UNIVERSITY OF OSLO
Department of Informatics

Improving the Physical and MAC layer models of NEMAN

Master thesis

Jan Erik Johnsen

May 24, 2006

Abstract

Mobile ad-hoc networks (MANETs) are wireless networks consisting of autonomous nodes without any infrastructure or central controlling entity. Such networks are highly dynamic and unstable, and prone to many problems such as traffic collisions, hidden nodes, noise, communication gray zones, obstacles and multi-path fading. Application and routing protocol development for MANETs is a difficult task due to these issues. They can have severe effects on the performance of a routing protocol or an application. It is necessary to test and evaluate MANET applications and routing protocols extensively in simulators and emulators, before deciding to go further by doing more expensive tests with actual equipment. NEMAN is an emulator for MANETs that can be used for such testing, but the original version lacks support for the physical and MAC layer issues that exist in real MANETs. This master thesis investigates some of the issues and problems that exist in MANETs, and how their effects might be simulated in NEMAN, making the emulator more realistic. Simulation of traffic collisions, communication gray zones, and random packet loss is examined and implemented in this thesis, improving the physical and MAC layer models of NEMAN. Functionality tests show that the implementation works as expected. The performance of the new NEMAN version is measured to be satisfactory, as it does not limit the total number of nodes that NEMAN can reliably emulate.

Preface

This thesis is part of my Master Degree at the University of Oslo, Department of Informatics. The work was carried out for the Distributed Multimedia Systems (DMMS) group, during the spring semester of 2006. My teaching supervisors were Professor Thomas Plagemann, the leader of the DMMS group, and Matija Pužar, the author of NEMAN. I would like to thank them for their guidance, help and valuable feedback throughout the period. I would also like to thank Bjarne Johnsen and Erlend Birkedal for proof-reading the report, and Erlend Birkedal for providing valuable help during the process.

Jan Erik Johnsen
May 2006

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem description and the goal for the thesis

A mobile ad-hoc network, also known as a MANET, is a wireless network consisting of mobile nodes where the following general characteristics apply [7]:

- **Autonomous nodes** - Each node is autonomous, and there is no central controlling entity in the network.
- **No infrastructure** - Communication is done directly from node to node, or via other nodes in a multi-hop fashion. Each node runs a routing protocol designed specifically for MANETs. The routing protocol discovers routes throughout the network, usually by using broadcasts to other nodes in radio range. Some nodes might provide gateways to a fixed network, but such a fixed network is not part of the MANET itself.
- **Dynamic topologies** - The radio range of a wireless node is limited, and nodes in a MANET are free to move arbitrarily. Thus, the network topology may change randomly and rapidly at unpredictable times. Each node is also free to join and leave the network as it pleases.
- **Bandwidth-constrained links** - Wireless links provide significantly lower capacity than their wired counterparts. The effects of multiple access, traffic collisions, fading, noise, interference, and etc, also lessen the actual throughput of a wireless link. Wireless communication is getting faster and faster as time progresses, but the bandwidth demands from the users are also increasing.
- **Energy-constrained operation** - Most mobile nodes depend on batteries for their energy, which means all operations on the node should be designed for energy conservation.

- **Limited physical security** - Wireless networks are generally more prone to physical security threats than wired networks due to the increased possibility of eavesdropping, spoofing, and denial-of-service attacks.

As we see, problems are the norm rather than the exception in MANETs. The nature of MANETs makes the process of developing good applications and routing protocols for these networks a difficult task. Given the limitations of MANETs, it is important to make the network as efficient as possible. A network emulator can be a helpful tool for evaluating and testing routing protocols and applications, without developing an expensive testbed consisting of real devices. An emulator makes it appear to the application that it is running on a real MANET device, while it is in fact running within an emulator on a standard PC. The main idea is that minimal changes should be needed to be done to a MANET application developed within an emulator, in order to make it work on a real device, and vice versa. Many MANET emulators exist today and NEMAN (A Network Emulator for Mobile Ad-Hoc Networks) [20], is one of them. NEMAN differs from other MANET emulators in that it can emulate hundreds of nodes on one single PC. For details on NEMAN, please refer to Sect. 2.2.2.

Wireless communication introduces many problems that MANET application developers have to address. Traffic collisions, hidden nodes, multi-path fading, communication gray zones, and so on, are some of the issues that are encountered on the physical and the MAC layer of wireless networks. The original NEMAN does not simulate such effects, but rather forwards data packets based solely on connectivity between nodes. This is not very realistic, as it emulates a MANET where all the links are perfect, and every node can communicate at the same time, without worrying about interference between their transmissions. The goal of this thesis is to make NEMAN slightly more realistic by improving the physical and MAC layer models of NEMAN through simulation of some of these effects and issues. Specifically, simulation of traffic collisions, communication gray zones, and random packet loss are examined and implemented in this thesis.

The main scope of the thesis has been on the design and actual implementation of these issues. A more thorough literature study on different models for the amount of random packet loss, or the transmit range of the IEEE 802.11 data rates, was of lower priority. Thus, the values used for this in the thesis are temporary values quickly determined for the sake of our testing, and not necessarily realistic values that will be used with the emulator in the future. A more thorough literature study on other models for random packet loss, and possibly the determination of the transmit ranges of IEEE 802.11b, are deferred to future work.

1.2 Terminology

In this report, the term *simulation* is used to describe the simulation of the specific wireless issues and the modeling of the wireless channel, while *emulation* is used more generally about the whole process of using a network emulator, with focus on the processes running on top of it. Furthermore, *simulation scheme* describes the implementation of the simulation of each issue. For the difference between a network simulator and a network emulator, please refer to Sect. 2.2.1.

1.3 Approach and method

The methodology for this thesis can be divided into the following steps:

- **Literature study** - Reading mainly research papers on the different physical and MAC layer issues of MANETs, as well as other MANET emulators.
- **Code study** - Source code study and experiments with the original NEMAN in order to get familiar with the emulator and the way it works.
- **Design** - Development of a rough design and some pseudo code for how the issues could be simulated in NEMAN. During this phase it was also decided which issues that were prioritized in the thesis.
- **Implementation and preliminary testing** - Each of the three issues were implemented and tested separately after each other.
- **Final evaluation** - Final, more thorough evaluation, as documented in this report.

All the steps included documentation and work on the report. The organization of the report reflects the methodology used for the thesis, as we see in the next section.

1.4 Organization of the report

- Chapter 2 gives a background on the different issues that are encountered on the physical and the MAC layer of wireless networks. Network emulation and NEMAN itself are also examined in this chapter.
- Chapter 3 presents the design for how the issues described in chapter 2 might be introduced into NEMAN.
- Chapter 4 contains the implementation details for the issues that are implemented during this thesis. The chapter also contains experiments with two configurable variables.

- Chapter 5 gives the evaluation of the new NEMAN in terms of functionality and performance.
- Chapter 6 concludes the report and future work on NEMAN is discussed.

Related work on the topic is referenced throughout chapter 2 and 3, where appropriate. A CD with the source code for the new NEMAN and more, is attached with the report. A description of the contents on the CD is given in Appendix A.

Chapter 2

Background

The background chapter consists of two parts. Sect. 2.1 investigates some of the physical and MAC layer issues that might be interesting to implement into NEMAN, while MANET emulation in general and the original version of NEMAN is described in Sect. 2.2.

2.1 Physical and MAC layer issues in MANETs

Sect. 2.1.1 discusses the signal strength of nodes in wireless networks. A very common problem are traffic collisions which occur when nodes within reach of each other transmit data on the same frequency at the same time. We discuss this in Sect. 2.1.2. One of the most important reasons for traffic collisions is the hidden node problem, which we investigate in Sect. 2.1.3. In Sect. 2.1.4, we describe communication gray zones, an issue that makes proper route discovery in mobile ad-hoc networks even more challenging. Furthermore, Sect. 2.1.5 describes some of the reasons for random transmission errors. Finally, wireless communication also suffers from delays, which is briefly described in Sect. 2.1.6.

2.1.1 Signal strength

The transmission power and receiving sensitivity of a wireless data transmission varies from node to node in a MANET, and it differs with the radio technology used. In this thesis we assume that the IEEE 802.11 standards are used, as these are the main wireless standards today. The radio waves pass through air, and the signal strength falls off sharply with the distance from the source [14]. Each receiving node has a signal power threshold. If the signal power is below the threshold, the signal will be ignored as noise. This means that the connectivity in MANETs is incomplete: a node cannot always send to all the other nodes in the same network. This is in contrast to

the complete connectivity of Ethernet where any node can send to all the other nodes connected to the network.

The received power can be modeled by the free space model and the two-ray ground reflection model as a deterministic function of distance [5], like the authors of MobiNet did [16]. When the receiving node is within the reference distance (typically around 100 meters) of the sending node, the signal degrades as $\frac{1}{r^2}$ according to the free space model, where it represents the distance. Beyond this reference distance, the signal degrades as $\frac{1}{r^4}$ according to the two-ray ground reflection model. The received power of a data packet transmission can be compared to the carrier-sense threshold and the receive threshold. Packets with values below the carrier-sense threshold are discarded as noise, and those above the carrier-sense threshold, but below the receive threshold, are discarded as error.

Radio signals typically have a very long range in free air, but the range is severely limited as soon as a physical obstacle like a wall is encountered. How much the signal is weakened depends on the material and the thickness of the obstacle. Physical obstacles also cause the multi-path problem, due to the fact that the transmitted signal takes different paths to the receiver when obstacles are encountered [26]. The received signal is composed of the sum of the various contributions, each of which differs in both amplitude and phase. These different contributions often interfere in a destructive manner that seriously weakens the signal strength. In some cases, multi-path causes fading as well, which consists of rapid changes in a radio signal's amplitude over a short period of time or travel distance, possibly destroying the transmission.

2.1.2 Traffic collisions

A traffic collision in a wireless network occurs when two or more nodes transmit on a radio channel during the same time frame. The different transmissions will most likely interfere and bit errors will be introduced to parts or all of the data, which then becomes useless. If two packets overlap at the receiver the probability of them interfering is big if the difference in power level of the two is smaller than 10 dB [16]. If the difference is bigger than 10 dB, chances are good that the strongest of the two transmissions will be received correctly, while the other will be lost.

In traditional IEEE 802.3 wired Ethernet, a scheme known as Carrier Sense Multiple Access with Collision Detection (CSMA/CD) is used to handle traffic collisions. The algorithm is simple but effective, and works as follows: A node transmits if it senses that the wire is idle. If it detects a collision while transmitting, it continues to transmit until the minimum packet time is reached, to ensure that all other transmitters and receivers also detect the collision. It then waits until the wire becomes idle again, before retrying after a random time. If there are more than one failed transmission attempts,

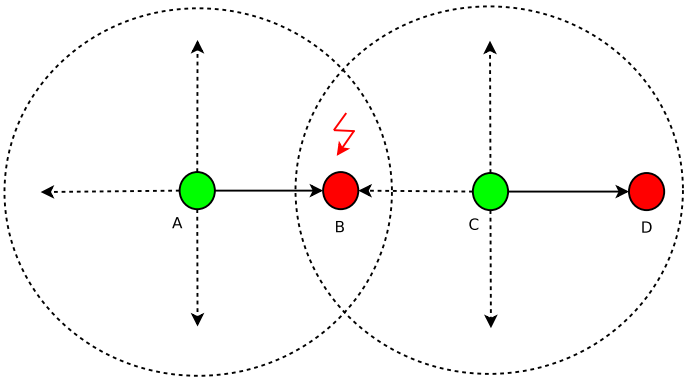

Figure 2.1: Example of the hidden node problem

the node uses an exponentially increasing back-off time before trying again [13].

CSMA works well for wired networks where reliable collision detection is possible, but for wireless networks this is not always the case, as we will see in the next section. The main wireless standards today are the IEEE 802.11b and IEEE 802.11g standards, operating at a maximum speed of 11 and 54 Mbps, respectively. These standards employ a technique known as Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). The main difference between this and CSMA/CD, is that rather than trying to detect collisions, which is not always possible, it tries to avoid the collisions from occurring in the first place. This is done by using a technique that we explain in the next section.

2.1.3 Hidden nodes

The hidden node problem arises if two or more nodes cannot "see" or "hear" each other, but their transmission ranges overlap. If two nodes sense the channel to be idle, and then transmit during the same time frame to the same receiving node that happens to be located in the overlapping area, their data transmissions will interfere at the receiving node. Consider the following example, illustrated in Fig. 2.1: In this topology, node C cannot receive node A's transmissions. If node A transmits to node B during the same time frame that node C transmits to D, the two transmissions will collide at node B. This means that node B will be unable to receive the message from node A. Note that the message from C to D is unaffected [22]. Seen from node A, node C is a hidden node. The hidden node problem is a case where reliable collision detection, like in Ethernet, is not possible.

The hidden node problem can severely affect the performance of wireless networks, especially when it comes to TCP and retransmissions over multi-hop routes. It is considered by the author of [23] to be one of the main challenges in future wireless architectures.

In order to mitigate the hidden node problem a variant of CSMA called MACA

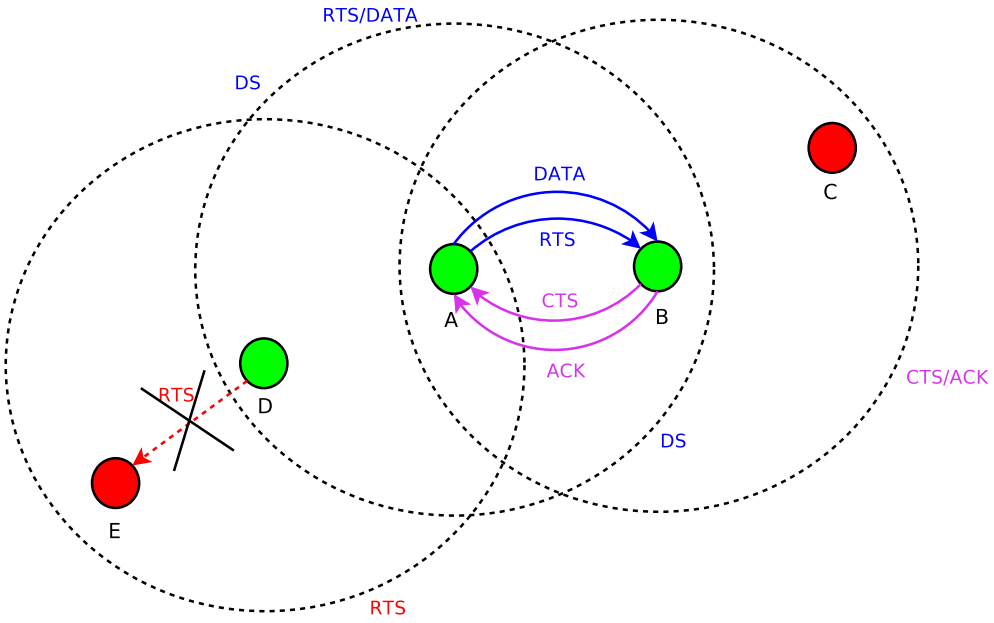

Figure 2.2: Example of data exchange using RTS-CTS-DS-DATA-ACK to mitigate the hidden node problem

(Multiple Access with Collision Avoidance) was developed. The initial proposal for this protocol was made by Phil Karn in 1990 [11]. In 1994, the authors of [3] proposed MACAW (MACA for Wireless). In order to better understand the hidden node problem and how it can be handled, we now describe the MACAW hidden node mitigation proposal.

The MACAW technique for hidden node mitigation is known as RTS-CTS-DS-DATA-ACK (Request-to-Send, Clear-to-Send, Data-Sending, DATA, Acknowledge), or more commonly just RTS/CTS. Let's start with a scenario example, as illustrated in Fig. 2.2: If node A wants to send data to node B, it first sends out an RTS-packet that contains the length of the proposed data transmission to B. If B hears the RTS, it replies with a CTS-packet that also contains the length of the data transmission. When node A receives the CTS-packet, it lets all other non-participating nodes in range, like D, know that the RTS-CTS exchange was successful by sending a DS-packet. Node A then starts sending the data to B. After B has received all the data, it sends an ACK-packet to A. The ACK-packet was introduced mainly to avoid having to recover errors at the transport layer when doing reliable data transfer, as TCP implementations typically have a minimum timeout period of 0.5 seconds. Using ACK at the link layer gives faster recovery as the timeout periods can be tailored to fit the short time scales of the medium.

Other nodes "overhearing" an RTS, like D, will not transmit anything during the time required for A to receive the CTS from B. Other nodes overhearing a CTS, like C, will not transmit anything during the time required to transmit the actual data, (as the length of the transmission was included in the RTS/CTS packages). This ensures

that nodes like C will not interfere with the data B receives, and nodes like D will not interfere with the CTS from B to A. In contrast to CSMA, this enables nodes to avoid collision at the receivers, and not the senders.

If D wants to transmit data to another node E, while A is transmitting to B, there will be problems as D will not be able to hear the CTS reply from E while A is transmitting. Thus, D cannot transmit any data. If it did, it would just retransmit the same RTS using longer and longer back-offs, like in CSMA, as it never receives the CTS from E. This is why node A must transmit a DS packet, to let D (and possibly others) know that it is actually transmitting data. Without the DS-packet, node D has no way of knowing whether the RTS-CTS exchange was successful or not, as it only heard the RTS from node A, and not the CTS from node B. If node D had not known that A was actually transmitting, it would not know when to start its transmissions to E. After the ACK packet time slot has passed, D can safely transmit its RTS to E.

CSMA/CA, used in the IEEE 802.11 standards, is a variant of MACAW along with CSMA [21]. Instead of sending the DS-packet, carrier sensing is used. All nodes use carrier sensing to check that the channel is idle before transmitting an RTS. Any node that overhears an RTS or a CTS is prohibited from transmitting any signal, i.e. they are blocked. This approach works well for infrastructure wireless networks, but [21] shows that the techniques used in IEEE 802.11 introduces many new issues for wireless ad-hoc networks. They argue that the blocked nodes under-utilize the network capacity. They also show how nodes might falsely block and how the effect of this can propagate through the network, sometimes even resulting in pseudo-deadlock situations. The RTS/CTS techniques also incur significant overhead and are often not used [22]. The use of the RTS/CTS mechanism is optional in the IEEE 802.11 standards. As we see the hidden node problem remains an important issue for mobile ad-hoc networks.

2.1.4 Communication gray zones

Routing protocols running on nodes in a mobile ad-hoc network use broadcasting of HELLO messages to discover their neighbors and thereby discover routes throughout the network. Broadcasting is done at a basic bit rate, 1 Mbps in IEEE 802.11, while unicast of data packets is done at the highest possible speed. Data sent at lower bit rates has a longer transmission range than data sent at higher bit rates. A communication gray zone occurs when two nodes are at the transmission borderline. Broadcast HELLO messages can be exchanged between the nodes in the gray zone, while unicast data messages can not [15], see Fig. 2.3. In the figure, node A can reach node B when broadcasting HELLO messages, but not when sending data, as node B is in the gray zone. This means that routing protocols using broadcast HELLO messages might give invalid routes if nodes are in each other's gray zone. Invalid routes lead to disruptive

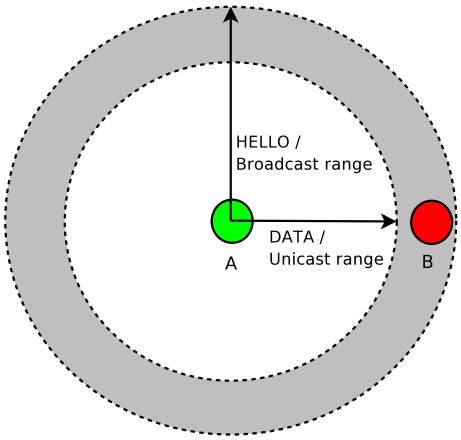

Figure 2.3: Example of a communication gray zone

behavior for applications with continuous packet flow, like streaming multi-media and large file transfers. The authors of [15] found that their Ad-hoc On-demand Distance Vector Routing (AODV) [19] implementation, AODV-UU [2], had an unexpected high amount of packet loss, especially during route changes. They found that the increased amount of packet loss coincided with the gray zones.

They identified four conditions that contribute to the formation of communication gray zones:

- **Different Transmit Rates** - In IEEE 802.11b, broadcasting is done at a basic bit rate while data transmissions use higher rates. Low rate transmissions are more reliable and can reach further.
- **No Acknowledgments** - In IEEE 802.11b, broadcasting is done without acknowledgments, which means that receiving a HELLO message does not guarantee that transmissions are possible in the opposite direction.
- **Small Packet Size** - The HELLO packets are small and thus less prone to bit errors than the larger data packets. This makes it more likely for a HELLO message to reach a receiver than a data packet, especially over weak links.
- **Fluctuating Links** - At the transmission borderline, communication tends to be unreliable due to fluctuating quality of links. One successful HELLO message does not reflect correctly whether consistent communication between two nodes is possible or not.

To address the problem of unidirectional links in AODV-UU they proposed that nodes exchange their neighbor set in an extension field of the HELLO messages. A node can then tell if the link is bidirectional by looking for its own address in the neighbor set. They also proposed to extend AODV-UU to use N-Consecutive HELLOs, where N is typically 2 or 3, to address the problem of fluctuating links. This

means that a node should only accept another node as a neighbor if it has received N HELLO messages. Their last proposal was to introduce a signal strength threshold on the AODV control messages so that the ones with a signal quality below a given threshold are discarded. This counteracts the gray zone problem by not using routes where link quality is so bad that only HELLO messages and no data can get through. It also minimizes the probability of unidirectional links. Their tests showed that the signal quality threshold approach almost totally eliminates the effect of communication gray zones. They were only able to obtain real world performance figures that matched the simulation results by enabling these modifications [15].

2.1.5 Random packet loss - Frame Error Rate

All wireless communication is subject to more or less random packet loss. This loss is often referred to as the Frame Error Rate (FER), which is the number of frames received with errors, divided by the total number of received frames [14]. The most important cause for the error frames, is probably noise and interference from other radio signals, as well as signal degradation due to distance. The amount of packet loss increases exponentially with the distance. Obstacles play an important role as well, especially since they cause multi-path fading, as previously mentioned, which in turn might destroy other previously unaffected signals.

2.1.6 Wireless delays

In wireless communication, a packet can be delayed for many reasons. The most important ones are radio wave transmission delay in the air, collision delay, CSMA/CA delay and the underlying queue delay [14]. Radio waves propagating through the air might take different paths, as well as encounter physical objects that alter their direction, causing the signal to be slightly delayed. CSMA/CA delays are caused by the collision avoidance techniques that delay packets while carrier sensing the channel, and possibly exchanging RTS/CTS packets before the actual data is sent. This might in turn cause queuing of the packets. For NE [14], the authors modeled these delays by using an uniform delay model that applied a constant delay on all packet transmissions.

2.2 MANET emulation

This section starts by investigating MANET emulation in general, in Sect. 2.2.1. The original version of NEMAN is then described in Sect. 2.2.2. Finally, the ns-2 scenario file format is described in Sect. 2.2.3.

2.2.1 MANET emulation in general

Generally, there are four different techniques that can be applied for analysis and comparison of protocols and algorithms in MANETs: analytical modeling, network simulation, network emulation, and real-world experiments [12]. Network simulation employs an abstract model of the network layers that usually gives a detailed model of the lower layers. The detail of the model usually depends on the goal of the simulator. Network simulation is often a trade-off between detailed simulation of the physical layer and the size of the simulated network: accuracy versus scalability. The Network Simulator (ns-2) [18] is a very popular network simulator. The main problem with network simulators, like ns-2, is that code used with them often needs to be completely rewritten before it can be used on real devices.

A real-world experiment is done on physical devices and thus provides the highest possible realism. The problem is that such test beds tend to be very expensive both in terms of hardware costs and human resources when doing an experiment. Therefore, this method has little scalability. It is also harder to get reproducible results in an real-world experiment, e.g. with respect to node mobility.

Network emulation is a good trade-off between an expensive real-world experiment and a simulator. It provides a hybrid approach, in which real layers are combined with simulation layers, using an emulation layer for encapsulation of the lower layers. This means that real code can be run directly above an simulated wireless network at the lower layers, which saves time, effort and costs when switching from emulation to a real-world test bed. An emulator can also take into account device specific limitations, e.g. processing power, buffer size and interrupt delays. Many emulation approaches allow good control of the effects of the wireless layer, which enables repeatability of experiments. Testing and debugging of code is also easier in such an emulation environment, compared to a real-world test bed.

A MANET emulator generally consists of the following components: scenario description, scenario generator, scenario parser, mobility generator, emulator core, GUI, routing daemon(s), and the program code that is to be run with the emulator. The scenario generator is usually a program or a script that generates a description of the scenario. The scenario description is an important part of the emulation, as it allows the reproducibility of parts of, or possibly the whole experiment. A very popular scenario file format is the ns-2 format, which we examine in Sect. 2.2.3. The scenario parser and the mobility generator use the scenario file to simulate node movement, and this information is used by the emulator core. The emulator core decides what network traffic gets forwarded through the virtual network, and when it arrives to its destination, by using a model of the wireless channel. This model consists of simulation schemes like traffic collisions, packet loss, wireless MAC layer emulation, and physical obstacles.

The most important role of the emulator core is to apply an encapsulation layer above the MAC and physical layer of the network stack. This way it appears to the program code and possibly the routing daemons, that they are running on a real MANET. The routing daemon(s) implementations are ideally done by a third party, and compliant to existing standards, like RFCs from IETF, with little or no modifications done to them in order to work with the emulator. This way the routing daemon implementation can be used with the emulator the same way as it would be used on a real device. However, this is not always possible. The authors of the emulator may be forced to reimplement or write their own version of the routing protocol. A GUI is usually also a part of the emulator, visualizing the node mobility and the virtual network, as well as allowing control of the emulator.

Different MANET emulators have different goals and approaches. Six different evaluation criteria for MANET test beds were introduced by [12], which is a survey on real-world and emulation testbeds for MANETs:

- **Scalability/Size** - addresses the number of nodes the test bed or emulation approach can support.
- **Development Simplicity** - addresses the complexity of developing the test bed regarding human resources for hardware manufacturing, software coding, or porting software to the platform.
- **Management Simplicity** - addresses the required management capabilities in terms of human resources during a test run.
- **Wireless Modeling Accuracy** - reflects the fidelity of the applied modeling approach for the wireless media.
- **Mobility Modeling Accuracy** - addresses the reproducibility of mobility. Nodes in the test bed may not be able to perform arbitrary mobility patterns or have strong constraints regarding velocity variations.
- **Cost efficiency** - in terms of cost for the required hardware and software, as well as the required space for deployment.

An example of a MANET emulator is MobiNet [16]. MobiNet is a hybrid emulation testbed, where multiple physical nodes emulate multiple virtual nodes. In addition to this, MobiNet employs one or more centralized core control nodes. All the traffic from the emulated nodes is forwarded to the control node, which simulates the layers below the network layer. The control node is configured to use certain emulation modules, e.g. a DSR routing module. Considering MobiNet against the evaluation criteria above, it is scalable, cost efficient, easy to manage, and mobility is reproducible [12].

NE [14] is another MANET emulator. NE is based upon wired Ethernet, where the traffic is shaped and the bandwidth is adapted to model the wireless channel. NE supports incomplete connectivity, random frame loss, bandwidth limitation, and communication delay. A central server broadcasts link information to NE clients running on other physical nodes in the same wired Ethernet. It is easy to develop and manage experiments with NE and mobility is reproducible, but it is not as scalable and cost efficient as MobiNet [12].

A more recent emulation testbed is TWINE: A Hybrid Emulation Testbed for Wireless Networks and Applications [27]. TWINE is very interesting as it allows both simulated, emulated and real nodes in the same testbed. Its goal is to combine the accuracy and realism of emulated and physical networks, with the scalability and repeatability of simulation, for evaluation of real protocols and applications. Thus, TWINE aims at fulfilling the scalability, mobility modeling accuracy and wireless modeling accuracy criteria.

2.2.2 NEMAN: A Network Emulator for Mobile Ad-Hoc Networks

NEMAN [20] is a MANET emulator designed to fulfill the following requirements:

- **Minimal initial effort** - Installing and using NEMAN should not consume too much time and effort, so that it can also be efficiently used in shorter projects.
- **Costs** - Low hardware and software costs, as well as human resources.
- **Scalability** - NEMAN should be able to run a high number of nodes without severe performance loss.
- **Portability** - The code developed for the applications and protocols that is to be used with NEMAN should be portable to genuine wireless devices with minor or no changes at all.
- **Realistic network layer** - A real implementation of a routing protocol should be used with NEMAN. The main concern when developing NEMAN was to reflect the effects of mobility in MANETs, i.e. the connectivity and loss of connectivity between nodes. Lower layer issues such as collisions in the air, hidden nodes, etc. were of lower importance.
- **Possible comparability** - NEMAN uses standard ns-2 scenario files for topology and node movement. This enables easier comparison with other similar tools, as ns-2 is used for much work in the area.

Comparing these requirements to the six evaluation criteria for MANET test beds by [12] in the previous section, we see that all except for *Wireless Modeling Accuracy* are

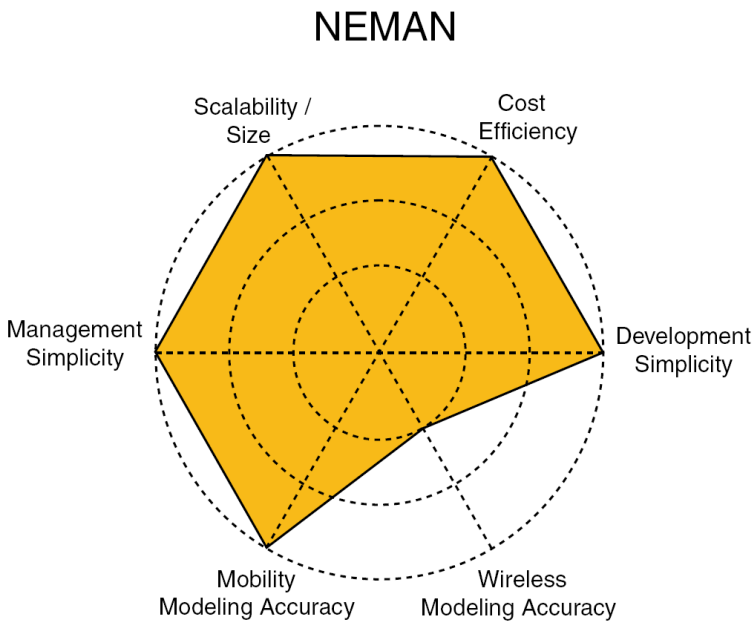

Figure 2.4: Evaluation of NEMAN using the criteria outlined in Sect. 2.2.1, figure borrowed from [12]

fulfilled by NEMAN. See Fig. 2.4 for an illustration on the criteria fulfillment. The goal of this thesis is to make the wireless simulation of NEMAN slightly more realistic, as this is what NEMAN needs the most. Everything in NEMAN runs on a single Operating System (OS) by using parallel virtual network stacks, which makes NEMAN a monolithic emulator. The only exception is the GUI, which is usually run on another computer, but it could just as well run on the same computer as the rest of NEMAN. The parallel virtual network stacks for the emulated nodes are realized through the use of TAP devices, which are the virtual Ethernet devices of Linux. Each emulated node corresponds to one TAP device. NEMAN creates the TAP devices, and user processes send and receive data via TAP interfaces using the standard socket application interface model. This provides the highest possible realism since the approach allows any application to run on any emulated node in the virtual topology. NEMAN captures the transmission requests, and only forwards the packets to TAP devices to which there is a direct link in the emulated network based on its current topology information. To make the mapping between the TAP interfaces and the nodes easier, each TAP interface has an IP address that corresponds to its node number. E.g.: 10.0.0.99 is the node number 99, 10.0.1.0 is node number 100, and 10.0.2.34 is node number 234.

By using this approach, NEMAN can emulate hundreds of nodes within a single physical computer. The only real bottleneck is the processing power of the computers used for the Topology Manager and the GUI. The GUI is the biggest bottleneck, as it is written in the scripting language Tcl/Tk. All components of NEMAN run in the user space of Linux. The goal of NEMAN is that for all protocols and applications it is to

appear like they are running on their own machine, but this does not always work with routing protocols. Routing protocols that are to be tested with NEMAN have to run as user space daemons, one daemon for each emulated node on each TAP interface. As there is more than one daemon, the kernel's routing table can not be used. This presents some problems if a routing protocol implementation is designed to make use of the kernel, as NEMAN does not use the kernel's routing table. When this is the case, the routing protocol implementation has to be changed not to use the kernel if it is to be used with NEMAN.

In a project assignment [4] in the course INF5090 [10], we encountered this problem when we tried to make the AODV-UU [2] implementation of AODV work with NEMAN. AODV is a reactive routing protocol, which means that routes are only discovered on demand. When a process tries to send data, the data is held back while route discovery is done. This means that the AODV-UU implementation depends upon its kernel-module to know when routes are to be discovered. As NEMAN needs to run multiple instances of the same routing daemon on a single OS, we had to remove the use of the kernel. This was done, but the AODV daemons still needed some sort of notification when data was being sent, so that route discovery could begin. Doing this would require major changes to both NEMAN and the AODV implementation, and was therefore not possible during the time-frame of the project.

For this thesis, NEMAN is used with the olsr.org [25] implementation of the Open Link State Routing protocol (OLSR), RFC3626 [6]. OLSR is a proactive routing protocol, which means that it discovers routes all the time, even when there is no traffic. As the protocol is proactive, it requires no communication with either the kernel or NEMAN to know when routes are to be discovered. This implementation works with NEMAN out of the box, though the route changes need to be reported to NEMAN through a separate script that parses the route change messages from the OLSR daemons.

The core of NEMAN is the Topology Manager, located in `topoman.c`. This application runs as a user space daemon. All packets sent to the TAP interfaces are delivered to the Topology Manager by the Linux kernel. It runs a loop that listens for incoming packets and processes them as they arrive. It also maintains the virtual topology by using a matrix that states whether each pair of nodes is in communication range or not. This link information is sent to the Topology Manager from the GUI. The GUI parses the ns-2 scenario files and sends the link information as the scenario moves along. The data packets sent through the virtual network via the TAP interfaces can be captured and monitored by using standard tools such as *tcpdump* on any of the TAP interfaces. The monitoring channel is one of the TAP interfaces, typically tap0. This enables the possibility to listen to the whole network on one place, as well as inducing traffic into the virtual network, as this interface has an open bidirectional connection to all the other TAP interfaces. See Fig. 2.5 for an overview of the NEMAN architecture. Fig. 2.6

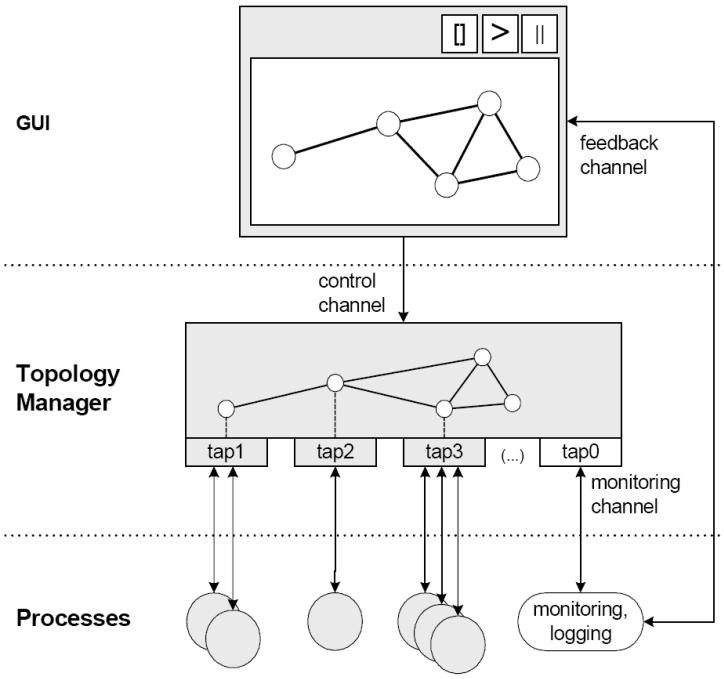

Figure 2.5: The NEMAN architecture, figure borrowed from [20]

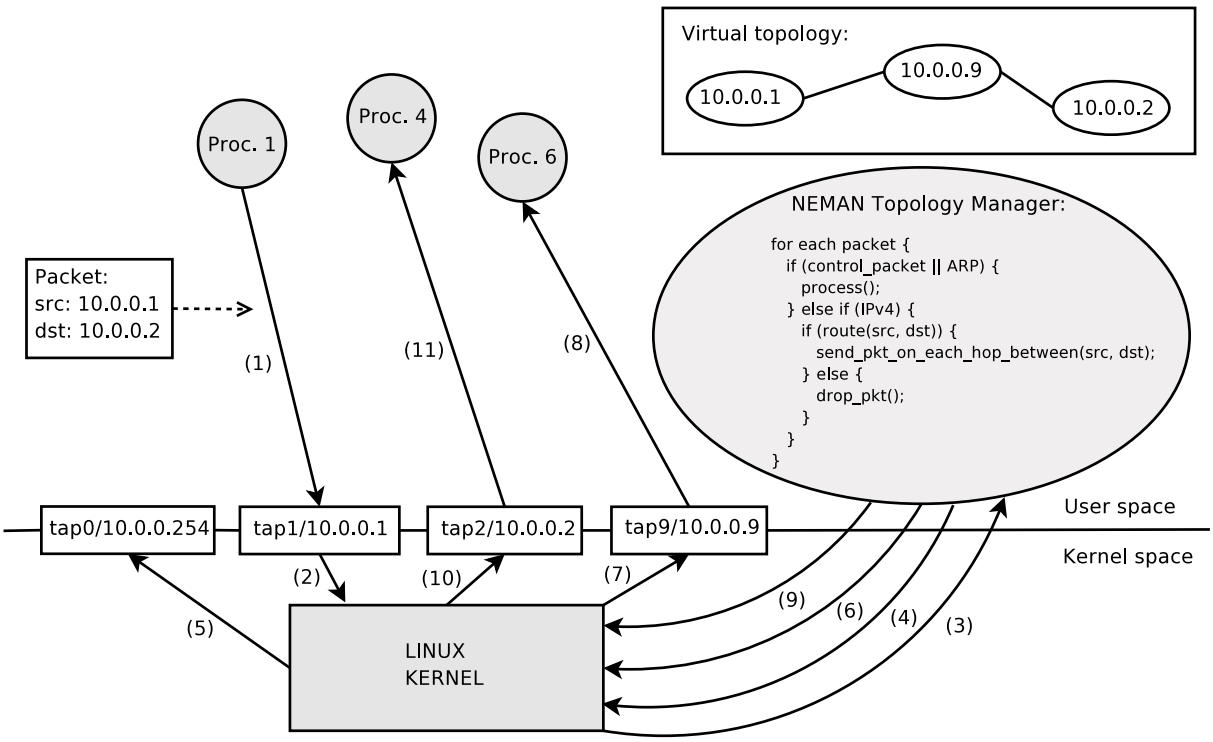

Figure 2.6: Example of the data flow generated from two nodes communicating in NEMAN

shows an example of the data flow generated between two nodes communicating in NEMAN, via an intermediate node. Here a packet is sent from the node with IP 10.0.0.1 to the node with IP 10.0.0.2, via node 10.0.0.9.

The NEMAN Topology Manager handles three different types of packets:

- **UDP control packets on port 3685 and 3686** - These are packets sent to the Topology Manager from the GUI and the script running the routing daemons on the TAP interfaces. There are 6 different commands supported by the Topology Manager: reset links, enable nodes, disable nodes, set link status, add/remove route, and the ability to turn of hop-by-hop forwarding for some nodes. Two commands can be sent to handle the routing daemons, namely start and stop.

- **ARP packets from processes on the TAP interfaces** - Currently, NEMAN handles ARP requests from the TAP Interfaces by replying to them without sending them to the actual receiver. NEMAN simply looks up the MAC addresses and switches the receiver and destination fields in the packets so that it seems to the requester that the answer came from the intended receiver. ARP is solved this way in NEMAN, because all local interfaces would otherwise respond with their own MAC address when they receive an ARP request, independently of the IP address that was queried.

- **IPv4 data packets from processes on the TAP interfaces** - These are the actual data packets from the virtual nodes that are to be sent through the virtual topology. If there is a route between the source and destination node, or if there is a direct link between them, the Topology Manager forwards the packet. For each hop it makes one packet for each direct link to make it appear like the packet is sent by radio to all the nodes in range (even if only one of the nodes is actually the intended destination for the packet). For multi-hop routes the Topology Manager also makes it appear like the packet is actually traversing the links in a hop-by-hop manner, by processing packet copies on each hop between the source and destination nodes. A user of NEMAN will be able to observe these packets on the monitoring channel (tap0).

In addition to the processing power bottleneck, running everything on one OS employing one CPU has two more disadvantages on the realism of the emulation; no two events really happen at the same time, and the scheduling of each event and task on the emulating machine is out of our control. Not knowing if two events happened at the same makes packet collision detection harder, which we describe shortly in Sect. 3.1. Arbitrary scheduling of the applications generating network traffic, the routing daemons, and NEMAN itself, means that an emulation experiment can never be repeated in the exact same way. This presents another evaluation criteria that is not covered by

the six criteria presented in [12], the repeatability or reproducibility of an emulation experiment. The criteria *reproducibility of mobility* only covers the actual movement of the nodes, which is fulfilled by NEMAN. The solution is to repeat NEMAN experiments and aggregate over the results when needed.

2.2.3 The ns-2 scenario file format

We now describe the ns-2 scenario file format, as this format is used by NEMAN and many other MANET emulators. Following is a simple example of a scenario with two nodes, generated by the ns-2 *setdest* program:

```
$node_(0) set X_ 444.27
$node_(0) set Y_ 120.50
$node_(0) set Z_ 0.00
$node_(1) set X_ 151.27
$node_(1) set Y_ 54.05
$node_(1) set Z_ 0.00
$god_ set-dist 0 1 16777215
$ns_ at 3.00 "$node_(0) setdest 292.31 166.91 0.92"
$ns_ at 3.00 "$node_(1) setdest 292.78 83.96 0.55"
$ns_ at 40.55 "$god_ set-dist 0 1 1"
$ns_ at 174.55 "$node_(0) setdest 292.31 166.91 0.00"
$ns_ at 177.55 "$node_(0) setdest 145.20 104.05 0.03"
$ns_ at 263.24 "$node_(1) setdest 292.78 83.96 0.00"
$ns_ at 266.24 "$node_(1) setdest 246.64 255.76 0.33"
```

The first six lines state the initial coordinates for each node in the scenario. Each node position is given with three coordinates, which means the scenario could be three dimensional, but as Z is 0 in this example, we only have two dimensions. The "set-dist" statement thereafter gives the distance between the two nodes at scenario start. 16777215 is used by *setdest* to indicate out of reach. The exact meaning of a distance of 1 depends upon the program used to generate the scenario file. *setdest* uses 250 units as the range of wireless transmitters, and only this, which means that a distance of 1 is 250 units or less, (in communication range). A distance value larger than 1 means that the nodes are out of reach. Another scenario file generator might use e.g. 2 to indicate a distance of 420 units, or anything else that might be appropriate. The rest of the scenario file states events that occur at the given timestamps, e.g. after 3.00 seconds for the first two events. The "set-dist" statement at time 40.55 seconds states that the distance between the two nodes changed to 1, that is 250 units or less. An "setdest" event changes the destination a node is moving towards, where the first two numbers

are the X and Y coordinates, and the last number is the speed of the node. The ns-2 format has support for many other statements, but the ones in this example are the only ones currently in use by NEMAN.

2.3 Summary

This chapter has given a background introduction to MANETs and we have looked at the most important issues that influence them. We have discussed MANET emulation, and the original NEMAN has been introduced. Based upon this information, we develop a design in the next chapter, in order to see how some of these MANET issues can be emulated by NEMAN.

Chapter 3

Design

NEMAN was developed mainly to emulate the connectivity and loss of connectivity between nodes in a MANET. Lower layer issues, such as those described in Sect. 2.1, were of lower importance. In this chapter, we investigate how some of the different issues might be implemented, adding to the realism of NEMAN. We start by describing the design for traffic collision simulation in Sect. 3.1, communication gray zones are then described in Sect. 3.2, and random packet loss in Sect. 3.3. Finally we also discuss different link speeds in Sect. 3.4 and CSMA/CA in Sect. 3.5, even though these two issues have not been implemented during this thesis.

3.1 Traffic collisions in NEMAN

Generally traffic collisions at receiving nodes can be modeled by looking for the fulfillment of **all** of the following conditions:

- **Overlapping transmission ranges** - Two or more nodes have overlapping transmission ranges.
- **Receiver in overlapping area** - The receiver of at least one of the transmissions is located in the overlapping transmission range.
- **Transmission is done at the same time** - At least two of the nodes having the overlapping ranges, with at least one receiving node in the overlapping range, are transmitting simultaneously.

Packets fulfilling these conditions collide at the receiver and should be dropped by NEMAN. This also covers the effect of the hidden node problem.

There are two problems when we want to do this in NEMAN. The first problem is that the Topology Manager currently has no space notion. Ideally this would be needed in order to detect packet collisions accurately. The problem can be solved two

ways. One is by looking at which nodes that have links between them, as a link is the equivalent of being in communication range. The other, and easiest way, is to simply check whether two or more packets sent at the same time are destined for the same receiver. The second problem is that all the packets that are to be sent via the TAP interfaces arrive to NEMAN as one queue of packets from the Linux kernel. Thus, NEMAN knows the absolute order of the packets, but not whether two packets were sent at the same time (and place), - which should have triggered a collision.

As an operating system on a single CPU machine may only run one application at a time, no two events occur at the same time. This means that we will never know for sure whether a collision between two or more packets would have occurred in reality. It is also the Linux kernel who decides which processes running on the TAP interfaces that are scheduled when, and consequently which packets that are to be processed first, determining the order in which they arrive to NEMAN. This scheduling is also done in an non-deterministic way, which means it will be hard to reproduce the same results for each emulation run.

Instead of knowing for sure what happened, we have to find a good and realistic approximation for how we might simulate that two packets were at the same place at the “same time”. As we know in real time when each packet arrived to the Topology Manager, we can time-stamp each packet and look at how much time elapsed between their arrival to NEMAN. We can then determine a constant that is used to determine whether two or more packets arrived close enough for a collision to occur, and use this as a window for possible packet collisions. We will refer to this constant as the Packet Collision Window (PCW). Please refer to Sect. 4.5 for a discussion on values for the PCW.

Combining this with the information we already have about the nodes that are in range of each other, provides us with what we need to determine traffic collisions. To sum up; for NEMAN, a collision at a receiver of a multi-hop packet is:

- **Two or more packets arriving at the same receiver** - The receiving node getting packets on two or more different links.
- **The packets are arriving at the "same time"** - The packets arrived at the receiving node within the Packet Collision Window.

By doing this, we are able to handle all the packet dropping locally in the Topology Manager, using functions operating on the incoming packet queue from the Linux kernel. Please refer to Sect. 4.2 for the details on how this was implemented.

An alternative to using a packet collision window, like the above proposal, could be to look at how much CPU time, both running and blocking, each of the processes sending data on the TAP interfaces have consumed. Monitoring this, NEMAN would know which processes have done something since the last check, and consequently

know that time, in the perspective of the process, has moved. Comparing the different CPU time usages from the processes would tell NEMAN which events happened at the "same time". This is a more complex solution, probably involving some kind of wrapper around every application on each TAP interface, to be able to measure the total CPU time of all applications on each TAP/node. This is unnecessarily complex, and it would probably take much more time to implement something like this as well, than just using a window. It would probably also make it more difficult to use NEMAN, as this wrapper would have to be applied around all the applications used with NEMAN in each experiment.

3.2 Communication gray zones in NEMAN

Gray zones can be simulated by dropping packets when the nodes are approaching their transmission borderline. The problem here is still the lack of space notion in NEMAN, as previously mentioned in Sect. 3.1. The NEMAN GUI reads ns-2 scenario files and determines which nodes are in communication range at any given time. This information is sent to the NEMAN Topology Manager as a binary value (connected or not connected), and the Topology Manager maintains a matrix of this information saying which nodes are able to communicate. This means that NEMAN does not know how close two nodes in communication range really are.

A possibility here is to change the NEMAN GUI so that it gives a more fine grained value for the distance between two nodes. The NEMAN GUI currently measures the distance as 1 and up, where 1 is close range. We could send this value to the Topology Manager instead of the binary one, and e.g. say that a distance greater than 5 means that the nodes are out of range. Using this measuring unit we could allow only broadcast messages (like the HELLO messages from the routing daemons), to get through at 5. In addition to this, we could introduce a certain success rate at which data/unicast packets are successfully received at the lower distances. As an example, 25% at distance 4, and from 3 and down we could make the success probability 100% for data packets. Broadcast packets would then have 100% success probability on all 5 distance steps.

It is also possible to differentiate between large and small packets, as larger packets are more prone to the gray zone problem. An example would be 10% success probability for large unicast packets and 25% for the small ones, at distance step 4. A large packet would probably be one around the MTU size, while a small one would be around the size of the headers plus a bit of data. The problem with a solution like this is that we have no data or measurements on how many packets that should be dropped at each step of distance, and it would probably vary greatly from network to network. Thus, we found it better to just differentiate between broadcast and data

packets, i.e. only broadcast packets are allowed at distance step 5 in the above example. This still gives us the effect of gray zones, as routes containing hops with a distance of 5 are established by the routing daemons, while data will not get through over these routes.

3.3 Random packet loss in NEMAN

General packet loss due to noise, obstacles, distance, etc, can be introduced by dropping packets randomly everywhere in the network. As previously mentioned, the amount of random packet loss is referred to as the Frame Error Rate (FER). If the outcome of a random number generator is to be used for the FER, it is important to test the outcome of the implementation carefully to determine that the results correspond well to the amount of loss we are aiming for. It might also prove useful to use a fixed seed for the random number generator so that the experiment can be run multiple times with the same simulated network conditions. However, the network conditions are never exactly the same, as the OS probably does not schedule everything exactly the same way each time an emulation is run, but it might still be useful as an option.

The distance between the sending and receiving nodes also affect the probability for random packet loss: the longer the distance, the higher the probability for packet loss. Thus, our implementation should also use the distance when determining if a packet is lost. This can easily be done when we know the distance steps, as described in the previous section.

For NE [14], the authors implemented a constant and an exponential model for the FER. In the constant model, their average FER was 0.02549, which means that 2-3% of all packets are lost. In their exponential model, the FER was given as $K(d - D_0)^{\beta} + P_0$, when the distance d is larger than the start distance for the model, namely D_0, which they typically set to 100 meters or less, see Fig. 3.1. Up to the distance of D_0 their model uses a constant FER, P_0, which they set to 0.0001. β is selected according to the target environment, where free space is given as 2, while a shadowed urban area lies in the range between 2.7 and 5. For an "open outdoor area", they used 3 in their example. Finally, K is a constant which they claimed to set to $2 * 10^7$, but they seem to have left out a minus, which means it should probably be $2 * 10^{-7}$, as this was the only way we were able to get the results from the model to match their example graph.

If we were to implement this model in NEMAN, we would have a problem with the distance values, as the current distance values supported by NEMAN are discrete, while a model like this requires a more continuous distance value. It is possible to get more continuous distance values in NEMAN by making the GUI send an update every time the distance between two nodes changes e.g. by one unit, but this would generate a huge amount of updates in a large scenario. This would result in increased process-

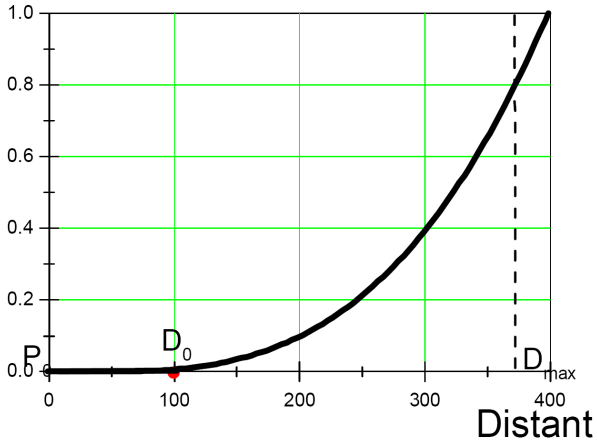

Figure 3.1: Example of the Frame Error Rate when using an exponential model, figure borrowed from [14]

ing on both the Topology Manager and the GUI, as well as increased network traffic between the two. As the main goal of NEMAN is to be able to run large scenarios, we should rather use the model to calculate a good FER for each distance step in advance, and use this FER for the entire step.

We should also make it possible to configure the distribution of the random packet loss over time, as [1] showed that the distribution of delivery probabilities varied greatly from link to link. One link would have really bursty packet loss in an on and off fashion, while another link in the same network would operate with a constant even loss rate, and still both links would lose the same amount of packets over the measured time frame. Thus, we should allow at least 2 different options for how random packet loss is done in NEMAN (bursty and constant).

An alternative, more advanced solution, might be to use the free space model and the two-ray ground reflection model together with the distance value to calculate the "signal strength" of each node. This "signal strength" value could then be used together with a constant FER that does not include the distance, to determine if a packet is lost or not. This way the success probability drops off gradually as the distance increases. The problem with this solution are still the discrete distance values in NEMAN. Basically, a solution like this is similar to using the exponential model from [14], as both solutions have low signal strength degradation and FER, respectively, before a reference distance of about 100 meters, and exponential increase after that.

3.4 Different link speeds in NEMAN

An important feature of wireless networks that significantly affects reliable communication between nodes is the link speed. In wireless networks like IEEE 802.11 that support different link speeds, the link speed is chosen after how well a given rate per-

forms. If e.g. 11 Mbps performs badly because the distance is too large, 5.5 Mbps or lower might still work over the same distance. This is known as Adaptive Rate Selection [9]. Lower data rates use less complex and more redundant methods for the data encoding, and are thus less susceptible to data corruption due to interference and signal attenuation. The general rule is that the slower you transmit, the longer the signal reaches (and more links become available). The experiments in [1] showed that there were about three times as many links at 1 Mbps as at 11 in Roofnet, a multi-hop IEEE 802.11 metropolitan area wireless network. As previously discussed, the different link speeds also contribute to communication gray zones, due to data transmissions using higher transmit rates than broadcasting [15].

Currently, NEMAN does not simulate link speed. It might be possible to simulate different transmit rates in NEMAN, e.g. by delaying packets that are supposed to traverse slower links, and making the transmit range of the nodes transmitting at lower bit rates longer than the rest. It would probably involve changing the scenario files in order to set the transmit range and speed of each node in a given scenario. Changing the scenario files is possible without breaking ns-2 compatibility if we put the transmit range and link speed as comments.

It also involves changes to the NEMAN GUI as it would have to parse this information from the scenario files, use it to determine the individual transmit range of each node, and send the link speed information to the Topology Manager. NEMAN would then use the link speed information to see how long it should delay the packet, and to determine for each node at which levels of distance communication is possible.

Another solution is to make the Topology Manager determine the link speed randomly for each node at startup, and then use the link speed to determine the delivery probability for borderline communication. This means that all nodes have the same maximum transmit range (as now), but the ones transmitting slowly are more likely to succeed when the distance is large. This would, however, make it impossible to reproduce the results. A good combination of the two solutions might be to only include link speed in the scenario files, and just let the GUI pass these values on to the Topology Manager.

Generally the range versus delivery probability is a clear trade-off, but only in free air where no obstacles are encountered. In a real wireless network where obstacles exist, the range versus delivery probability varies depending on the obstacles. [1] showed that transmit range versus delivery probability did not always correlate in Roofnet, most likely due to obstacles in the environment, different antenna heights and multi-path fading. This means that the gained realism from implementing different link speeds, and from that different transmit ranges, in the ways proposed here would be limited. Locking each node to one link speed would not be very realistic either, as a real wireless node would adapt its speed to the network conditions. Making

our virtual nodes do such an adaptation would not be possible without doing major changes to NEMAN, probably by implementation of parts of the IEEE 802.11 MAC layer protocol. This is not possible given the time-frame for this thesis.

We would also need to find proper data on the different transmit ranges for each transmit rate in IEEE 802.11, or at least a good model for it. In addition, we would have to find out how long each packet should be delayed, for each simulated link speed, in order to make the actual "speed" as close to reality as possible. This again would be dependent on the processing power of the computer running the Topology Manager. It would also be difficult, or maybe impossible, to hold back each packet for exactly a given time, when there are many nodes and much traffic. In other words, we can not guarantee that a packet still not ready for delivery, will be delivered in time, when the Topology Manager and the OS is busy doing other things concurrently. Much time might pass before a delivery thread is scheduled and able to check again if it is time to deliver the delayed packet. This might only work if the available processing power is significantly higher than the load on the computer, such that a delivery thread could busy wait while constantly checking its queue of delayed packets for packets that are ready for delivery. Due to these reasons, link speed simulation was not of priority in this thesis.

3.5 CSMA/CA in NEMAN

After traffic collision simulation and the hidden node problem are introduced into NEMAN, it would be interesting to implement the possibility to simulate CSMA/CA at the MAC layer of the nodes as well, as CSMA/CA is used in IEEE 802.11. Without using one of the MAC layer collision handling/avoidance protocols, the effect of the collisions in NEMAN will be much more severe than they would in a real wireless network that employs such a MAC layer protocol. Implementing CSMA/CA or similar, is too much for this thesis, but it is very interesting for future work on NEMAN.

3.6 Summary

In this chapter we have discussed how to implement traffic collisions, communication gray zones and random packet loss in NEMAN. These three issues were the ones prioritized and implemented during the thesis, and in the next chapter their implementation details are described. We have also discussed different link speed simulation and CSMA/CA, which is two issues not implemented in this thesis.

Chapter 4

Implementation

This chapter examines the implementation details of the changes made to NEMAN. The more basic changes to NEMAN are investigated in Sect. 4.1, as these are common to the three implemented simulation schemes: traffic collisions, communication gray zones and random packet loss. In Sect. 4.2 we describe the traffic collision simulation implementation, Sect. 4.3 describes the gray zone implementation, and random packet loss is described in Sect. 4.4. Finally we discuss and experiment with configurable parameters and values for the implementation. The PCW is discussed in Sect. 4.5, and `SLEEP_TIME` is discussed in Sect. 4.6.

4.1 Basic changes to NEMAN

The original NEMAN did not queue the packets, but rather processed and dispatched them one by one in a loop as they arrived. In order to be able to detect traffic collisions, NEMAN had to be changed to queue the packets instead, and run continuous processing on that queue in order to determine traffic collisions. This means that the forwarding is delayed slightly to allow the simulation to work through all the packets. This queue of the incoming packets from the Linux kernel is the central part of the solution for packet collision, gray zone and random packet loss simulation. NEMAN determines which packets are to be dropped from the queue by applying the chosen simulation schemes. We have made it possible to configure which simulation schemes are to be used by editing C constants in the top of `topoman.c`, so that the user of NEMAN can choose how the simulation is done.

We found that the best way to implement this was by dividing NEMAN into two processing threads, as this would allow NEMAN to handle its existing tasks concurrently with the newly added simulation schemes. The main thread sets up everything, starts the sub-thread, and runs the main loop that receives the packets from the Linux kernel. This works basically like the original NEMAN, except that the data packets

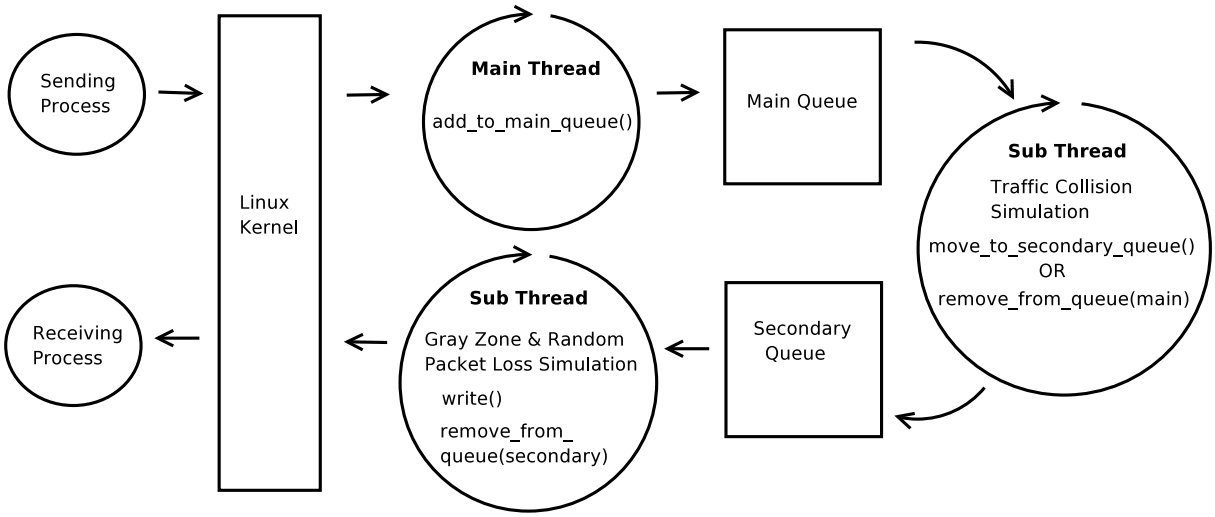

Figure 4.1: The data flow between the two threads and the two packet queues

from and to the TAP interfaces are added to the packet queue instead of being written directly to the receiving interfaces. The control packets for NEMAN and the ARP packets are still handled directly by the main thread.

The new sub-thread applies the various simulation schemes that might cause packets to be dropped. It determines for each packet whether it should be dropped, and puts the packets that pass the traffic collision simulation step into a second queue, see Fig. 4.1. This second queue was added to make it easier to differentiate between the packets that belong to the window currently being processed and the ones that arrived after the current window. It also allows greater concurrency, as the main queue is shared between the two threads, while the secondary queue is only accessed by the sub-thread. If collision simulation is disabled, *all* the packets in the main queue are added to the secondary queue, thus making that the current window. The determination of the window size is the reason why packet collision simulation is done directly in the main queue, and not the secondary queue, like the gray zone and random packet loss simulation. After possibly doing gray zone simulation and random packet loss, the packets left in the secondary queue are finally delivered to the receiving TAP interfaces.

This order of the simulation schemes means that packets lost due to communication gray zones or random packet loss might still collide with other packets sent at the "same" time and place, as collision detection is the first simulation scheme. This is the most correct execution order, as a packet transmission lost due to a gray zone or random packet loss is still present in the air, even though it did not make it to its intended destination. This means it can still interfere with other transmissions, thus giving a collision.

Following is a pseudo code overview of the sub-thread:

```
while (!safe_to_kill){

  if (main_queue_is_not_empty){

    set_current_window();

    if (collision_simulation_is_enabled){
      do_collision_simulation();
    }

    while (packets_in_main_queue_belonging_to_current_window){
      if (packet_has_been_lost){
        remove_from_queue(main_queue, packetID);
      } else {
        move_to_secondary_queue(packetID);
      }
    }

    if (gray_zone_simulation_is_enabled){
       do_gray_zone_simulation();
       if (packet_has_been_lost){
         remove_from_queue(secondary_queue, packetID);
      }
    }

    if (random_packet_loss_simulation_is_enabled){
      do_random_packet_loss_simulation();
      if (packet_has_been_lost){
        remove_from_queue(secondary_queue, packetID);
      }
    }

    for (each_packet_in_secondary_queue){
      write_packet_to_receiver(packetID);
      remove_from_queue(secondary_queue, packetID);
    }

  }
  usleep(SLEEP_TIME);
```

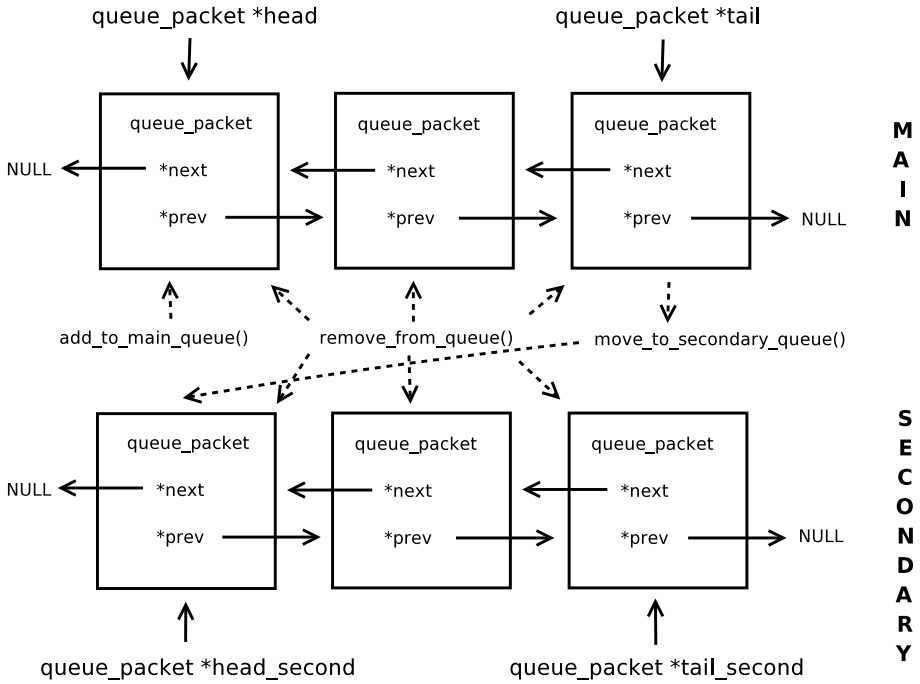

Figure 4.2: The two packet queues, their pointers, and the functions working on them

```
}
```

The queues are two linked lists of packet structs, where the packets are processed in FIFO[1] order, see Fig. 4.2. Each of the queues have a head and a tail pointer, where head points to the most recently added packet, and tail points to the oldest packet in the queue (the first one to be processed). The pointers `queue_packet *head` and `queue_packet *tail` are used for the main queue, while `queue_packet *head_second` and `queue_packet *tail_second` refer to the secondary queue. As the main queue is shared between the two threads, a mutex was introduced on the head and tail pointers for this queue to ensure consistency. This means that the two threads always call the function `void mainqueue_lock()` before reading or modifying the main queue pointers, and `void mainqueue_unlock()` when they are finished. The lock function is implemented using the pthread mutex locking function. The struct type `queue_packet`, used in the queues, is given below:

```
typedef struct packet {
    int           id;
    int           hopbyhop_id;
    int           source_node;
    int           dest_node;
    unsigned char buf[MAXPKTSIZE];
    int           length;
```

[1] First In First Out

```
    double        arrivaltime;
    int           lost;
    int           non_existent;
    int           bcast;
    int           actual_pkt;
    struct packet *prev;
    struct packet *next;
} queue_packet;
```

Each `queue_packet` has two distinct ID fields to identify them. The `id` field is unique for each packet, while the `hopbyhop_id` is the same for all the packets derived from one multi-hop packet. Remember from Sect. 2.2.2 that NEMAN makes n packets from each multi-hop packet, where n is the number of links the packet is supposed to traverse, plus tap0. This means that we need `hopbyhop_id` in order to determine which packets in the queue are in reality derived from the same packet. Using this we can easily locate and remove all the rest of these packets from hops after the current hop if the packet has been lost on the current hop. The `lost` and `non_existent` fields are used to indicate whether the packet has made it to its destination or not. Packets directly lost in simulation get a `lost` value of 1, while `non_existent` is set for multi-hop packets on hops after the hop where the packet was lost. By doing this, we can differentiate between the ones that actually were lost, and the ones that never were supposed to exist at all, as they were lost on a hop prior to this. We get back to this in Sect. 4.2.

The `source_node` and `dest_node` fields contain the node number of the sending and receiving nodes. These values are used by the sub-thread when the packet is written to its destination, and when checking for collisions, see Sect. 4.2 for details. The `buf[MAXPKTSIZE]` field is the actual data packet, already made ready for delivery by the main thread, and `length` is its length. The `arrivaltime` field contains the UNIX time-stamp for when the main thread fetched the packet from the Linux kernel, and is used by the collision detection simulation. It is stored in seconds and microseconds as a double with six digits after the decimal point. This has proven to be more than enough for telling the different arrival times apart on the test machine[2], even when packets are sent right after each other. This might, however, prove to be insufficient on faster computers in the future. An alternative is to store a pointer to the `timeval` struct instead, but then we would have to calculate the combined value of the seconds and the microseconds stored in this struct every time the arrival time is needed.

The `bcast` field has a value of 1 or 0, indicating whether the packet is a broadcast or unicast packet. This value is used by the gray zone simulation, see Sect. 4.3. The "actual packet" field, `actual_pkt`, also has a value of 1 or 0, and it indicates whether

[2] An Intel Pentium 4, 2.4GHz

this packet is the one bound for the actual intended receiver of the packet, or if it is just a copy generated to be sent to another node in radio range of the sending node. If the packet is a multi-hop packet, the one packet on each hop that is actually traversing the multi-hop route has this field set. This field makes it possible to determine whether a packet that just collided is a packet in a multi-hop route, which means we have to mark packets on later hops as `non_existent`, see Sect. 4.2 for more on this issue.

The pointers `*prev` and `*next` are used when traversing the linked list, enabling us to start in either end of the queues. The packet previously added to the queue is given by `*prev` as we move towards the tail of the queue, and the next packet added to the queue is given by `*next` as we move towards the head of the queue. Traversal of a linked list in one direction is often sufficient, but we need to be able to do both. An example of traversal using the `*next` pointers is when we are detecting traffic collisions. We have to start this check at the tail of the queue, as starting in the other end would mean looking for collisions between packets that are maybe not even meant to exist, since they collided on a hop prior to this one. On the other hand, we need the `*prev` pointer when we start at the head of the queue and move backwards to determine where the window should start.

There are three functions operating on the queues. The main thread adds the data packets to the main queue by calling the function `add_to_main_queue()` that takes information about the packet and a pointer to where the packet is stored as arguments. The function uses `malloc()` to allocate memory for the `queue_packet` struct, which means the only limit on the size of the queues are the available memory on the machine running the Topology Manager. The function `remove_from_queue()` removes the packet with ID `id` from the queue with the tail and head pointed to by `**queue_tail` and `**queue_head`. In order to avoid locking if we are removing from the secondary queue, as this queue is only accessed by the sub-thread, an additional integer argument called `main` tells the function whether the mutex function `mainqueue_lock()` is to be called before accessing the head and tail pointers. The function `move_to_secondary_queue()` moves the packet with ID `id` from the main to the secondary queue. Please refer to the sub-thread pseudo code above to see where in the sub-thread the queue functions are used.

4.2 Traffic collisions

We start this section by having a quick look at pseudo code for the traffic collision simulation in the sub-thread, after which we explain it in detail:

```
if (collision_simulation_is_enabled){
```

```
  while (window_has_not_passed) {
    usleep(SLEEP_TIME);
  }
  locate_where_in_the_queue_the_window_should_start();

  if (start_and_end_of_window_not_same_broadcast/hopbyhop_packet) {
    for (each_packet_in_window) {
      if (not_tap0 and not_non_existent) {
        for (each_packet_in_window_after_current) {
          if (current_collided_with_a_packet) {
            mark_both_packets_as_lost();
            if (actual_packet_collided) {
              for (each_packet_in_window_with_same_hopbyhop_id_ \
                   on_next_hop_after_colliding_packets) {
                mark_packet_as_non_existent();
              }
            }
          }
        }
      }
    }
  }
}
```

The traffic collision simulation starts by checking whether the PCW has passed. This is done by comparing the `arrivaltime` of the tail packet in the queue with the current time. If the window has not passed yet, the sub-thread sleeps for `SLEEP_TIME` microseconds, before checking again. It is important not to sleep too short, as this means the sub-thread is wasting CPU time on busy waiting, but it is also important not to wait too long, as that might result in the window passing long before the next test. `SLEEP_TIME` is discussed further in Sect. 4.6. The same value is also used in the test that checks the main queue for packets in the main loop of the sub-thread. If we wait too long, it might happen that packets that should not be in the window arrive before the current time check, but after the PCW, see Fig. 4.3. In this example, the window is set to start at packet #3, while it is more correct to have it starting at packet #2, as this is the last packet before the PCW value. Because of this, we have to check whether the window should start earlier in the queue, as indicated in the pseudo code. A window like this is called a jumping window, as the different windows are not overlapping each other (like a sliding window would).

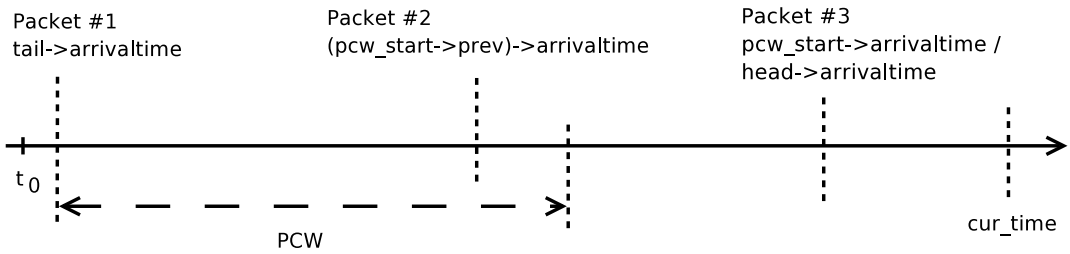

Figure 4.3: This example shows the PCW and pcw_start for three packets where pcw_start is packet #3, but should have been packet #2

After the window is set properly, the collision simulation checks that the first and last packet in the window are not the same. As discussed earlier in Sect. 2.2.2, the Topology Manager makes multiple packets for each packet, one for each hop in multi-hop routes, and one for each direct link on each hop (all nodes in range of the sender). All these packets derived from the same one are not supposed to collide with each other. Thus, we can skip the whole simulation if the first and last packets in the window are the "same one". We know this by looking at the packet's `hopbyhop_id` or `arrivaltime`, as packets that are the "same one" have the same same values in these fields.

For the multi-hop packets the fact that all the packets have the same time-stamp lower the realism somewhat. Ideally their time-stamps should have been different, and their forwarding on each hop should have been a little delayed, allowing other events in the network to get in between them. This is not done as it would be hard to gain any realism when everything is done on the same machine, which means no events are really at the same time, and the execution order of everything is not necessarily the "correct one". Thus, it would be hard to make any other events in the virtual network get in between the packets that traverse a multi-hop route, even if major changes were done to NEMAN. These changes would probably involve multiple threads that receive packets from the TAP interfaces to achieve higher concurrency.

After determining that the window is not just the "same packet", we enter a double for-loop where each packet in the window is checked for a collision with each packet after itself in the window. Packets to and from tap0 are not meant to traverse the virtual topology, and are therefore not checked for collisions. The tap0 packets still need to be going through the queues instead of being written directly, as we need to be able to remove them if it later turns out they were not supposed to exist (i.e. they were lost on a prior hop). We also check that the packet has not been marked `non_existent` in a previous iteration, as discussed in Sect. 4.1. If two packets in addition to this also have the same destination node, and different source nodes, they are found to collide. Two packets with the same source node should clearly not collide, as each node can only produce one packet at the same time. Both colliding packets are marked by setting the

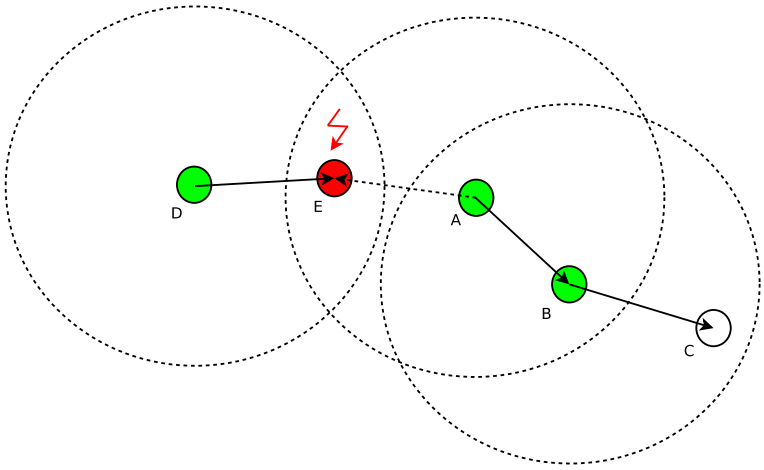

Figure 4.4: A sends multi-hop packet to C via B, this transmission collides with the packet from D to E at the receiving node E, but the multi-hop packet is not supposed to be affected by the collision

`lost` variable in their packet structs. Note that the `lost` variable enables us to continue checking the current window for other packets that might also have collided at the same receiver during the same window, and by that enables us to detect collisions between more than two packets at once.

Finally, we check in the double for loop for each of the colliding packets whether it was the "actual packet", that is, the packet bound for the actual intended receiver of the packet, and not just a copy generated to be sent to another node in radio range of the sending node. The loss of an "actual packet" means that there might be multi-hop packets for later hops after this packet in the queue. These are no longer meant to exist once the "actual packet" is lost on a prior hop. Thus, we have to go through all the other packets in the window that are the "same one", i.e. have the same `hopbyhop_id` as the two packets that collided. All these packets are located right after each other in the queue, and get marked as `non_existent`. We only want to mark packets on hops *after* the current hop as `non_existent`, as packets on the current hop might still reach other receivers in range, even though they collided at one of them. This means that the packets on the hop where a collision occurred are always sent to the monitoring channel, even though the intended receiver of the packet never received it. We have chosen to keep the tap0 packet since the packet that the tap0 packet was made from was actually transmitted. Even though it did not reach its destination, it was still present in the virtual network, and should thus appear on the monitoring channel.

It is important to check the "actual packet" field of the colliding packets, since we should not mark the packets of a multi-hop packet as `non_existent` unless it was the "actual packet" that collided. This is easily illustrated with an example, see Fig. 4.4. In this example, node A wants to send a multi-hop packet to node C via node B. The transmission from node A to B also reaches node E, where a collision occurs with a

transmission from node D to E. Here, the transmission from node A to B causes a collision, but the result is that since the "actual packet" from A to B is not the one that was lost, the packet from node B to C is not marked as `non_existent`.

For now, traffic collision simulation is not applied to ARP packets since these are handled differently, as explained in Sect. 2.2.2. A possibility might be to also put these in the queue with their requester as destination, and their intended replier as source, but it would not be very realistic as the replier never really sent the packet, and the initial request packets never traversed the virtual topology.

4.3 Communication gray zones

The gray zone simulation is implemented by differentiating between broadcast and data packets. As discussed in Sect. 2.1.4, broadcasting is done at a basic bit rate, while the data packets are sent at higher bit rates. This means that the broadcast packets have a greater transmit range than the data packets. A gray zone occurs when the routing daemons use broadcasting to determine connectivity between nodes, as a data exchange might fail even though the routing daemon has determined that a route exists. NEMAN does not implement different link speeds yet, but a good approach is still possible by using the distance between each set of nodes to define where the gray zone is located. If the distance is larger than a given value, NEMAN only allows broadcast packets to get through. This means that routes are established when data packets cannot be exchanged, giving us the effect of a communication gray zone.

We decided to use only two different transmit range steps; 0-250 meters for 11 Mbps, and 250-500 meters for 1 Mbps, which means our gray zone is located in the latter range. See Fig. 4.5 for an example of these transmit ranges where node A and node B are communicating while they are in each others gray zones. This means that only broadcast data might be sent when the distance is greater than 250 meters. We chose 250 meters as our 11 Mbps range, as this is the standard free air range used by ns-2 and many other simulation tools [24]. This also means that we are assuming that all nodes transmit data at their maximum link speed all the time, as different link speeds are not implemented in this project. For the 1 Mbps/broadcast range we had problems finding specifications on typical maximum values for the distance on IEEE 802.11b. The specifications for the Dell TrueMobile 1180 wireless card states 550 meters as its maximum transmit range at 1 Mbps in an open office environment [8]. Thus, 500 meters is a good enough rough estimate for now. In practice, the actual range of each wireless node varies with each wireless card and the transmit power used. This means there is no clear "correct way" of obtaining exact data on the transmit ranges, other than doing field experiments for an exact setup of cards and settings, and using these results. Even then, the results would also highly depend on the physical setup

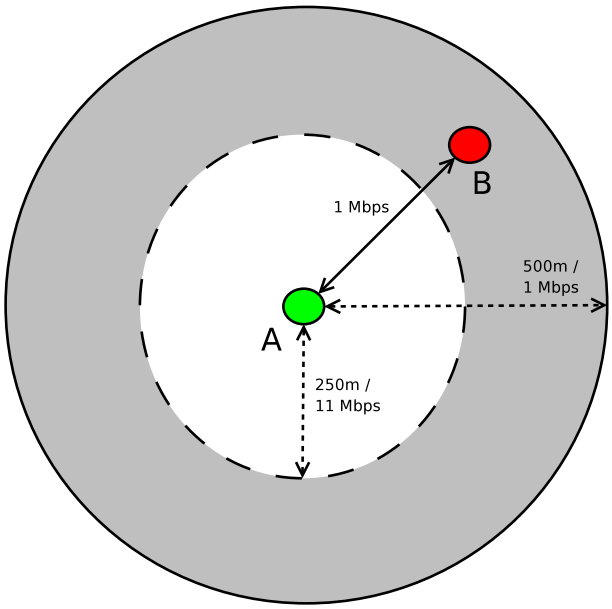

Figure 4.5: Example of the transmit ranges of 11 Mbps and 1 Mbps

of the landscape, obstacles, etc. This is out of the scope of this thesis, thus 250 and 500 meters are good enough general estimates for now. For future work, it might be interesting to determine the transmit ranges of the different IEEE 802.11b rates for a specific setup, either through literature study, or by doing a practical experiment, and use these values with NEMAN to make it more realistic.

As previously discussed in Sect. 2.2.3, the ns-2 scenario file format used by NEMAN contains the distance between node pairs given as a step value, e.g. 1 and 2, where 1 could be up to 100 units, 2 up to 200 units, and so on. When we use the ns-2 program *setdest* to generate random way-point scenario files, we have no control over the unit distance value for each of the steps described in the generated scenarios. As we want to use 0-250 meters and 250-500 meters as our steps, we have to change the scenario files so that they conform to these steps. To do this, the author of NEMAN wrote a script, *mkdist.pl*, that takes a ns-2 scenario file as input and changes it to use our chosen steps as distance values. Using one distance unit in the scenario files as one meter, we get the following distance values: **1** 0-250m, **2** 250-500m, and **3** for more than 500m/out of range.

Now the NEMAN GUI can determine each time the distance between a pair of nodes changes from one step to another, by reading the scenario file. Originally the GUI only sent information to the Topology Manager about whether two nodes were in reach or not. Now the distance step is sent instead, so that the Topology Manager can use this value for gray zone simulation. The GUI was also changed to display the distance between the nodes in different colors, where dark green represents the short links, and light green shows the longest links where the gray zone is located. See Fig. 4.6 for an example screenshot where the link between nodes 1 and 2 is under 250m

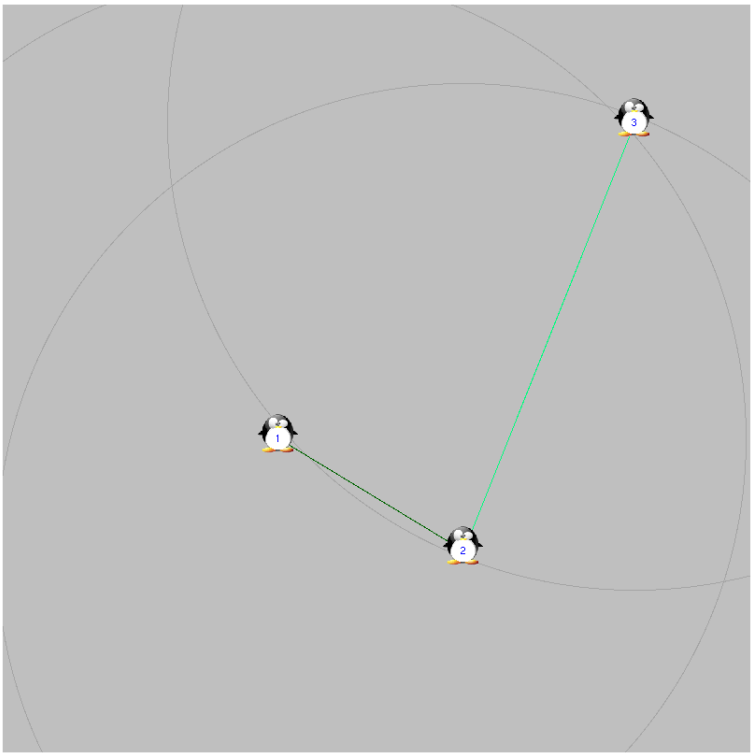

Figure 4.6: Example screenshot from the NEMAN GUI

long, while the link between nodes 2 and 3 is between 250 and 500m, which is in the gray zone range. Following is an pseudo code overview of the gray zone simulation:

```
if (gray_zone_simulation_is_enabled){

  for (each_packet_in_secondary_queue) {
    if (packet_is_not_broadcast
        && not_to_or_from_tap0
        && src_and_dst_node_is_in_gray_zone) {
      remove_non_existent_packets_after_gray_zone_packet();
      remove_gray_zone_packet();
    }
  }
}
```

The gray zone simulation can now simply check each packet in the secondary queue for packets that are not broadcast and at the same time sent over a link that has a distance value of 2, corresponding to the gray zone. We obviously do not want to check the tap0 packets, as these are not meant to traverse any links or get lost in simulation. If a packet is found to match the criteria, the Topology Manager proceeds to remove all the packets on later hops after the hop of the one that got lost, as these are not

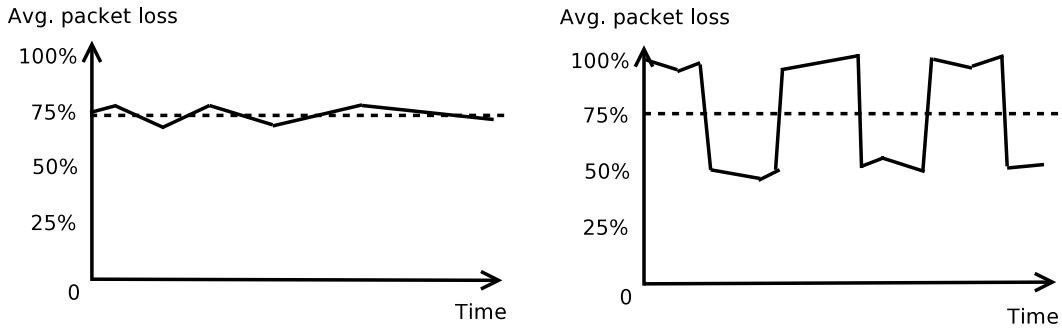

Figure 4.7: The figure shows an example of constant packet loss to the left, and bursty packet loss to the right

supposed to exist. Like discussed in the previous section, it is also important here that we only remove later hop packets if the later hop packet is actually the one lost in the gray zone, and not just a packet generated for another node in radio range. Finally the packet itself is removed. This solution might have one possible issue. In theory, the topology information used in the gray zone simulation might be different from how the virtual topology was when the packet was actually sent, as time progresses while packets are processed. This will not be a problem in practice, as the computer doing the gray zone simulation is processing it fast enough to make it unlikely that a change like this would occur in the virtual topology.

4.4 Random packet loss

Random packet loss is implemented to occur either constantly or in bursts. This is configured by the constant `RANDSIMU_BURSTY`. The idea is that the total amount of packet loss should be the same with either option. The constant loss is done by randomly removing packets at the same rate all the time. The bursty loss is 25% higher than the constant loss half of the time when a packet loss burst occurs, and 25% less than the constant loss the rest of the time, which means the total loss should be about the same. See Fig. 4.7 for an example of constant and bursty packet loss, where the average loss is 75% for both. In addition to this, the two distance steps introduced for the gray zone simulation, are used to differentiate, such that the probability for packet loss is significantly higher on 250-500 meters, compared to 0-250 meters. The seed for the random number generator is set by the constant `RANDSIMU_SEED`. By default we use the current time as seed, which means a different seed is used each time NEMAN is restarted.

The exact probability for packet loss on each level of distance is configurable through C constants. For the constant mode, the loss probability at the two distance steps is given by `RNDCONST_LINK1` and `RNDCONST_LINK2`, respectively. For the bursty mode `RNDBURST_HI_LINK`n, where n is the level of distance, gives the loss probability

when the bursty packet loss is high. Likewise, RNDBURST_LO_LINK*n* defines the loss when the bursty packet loss is low. We generate a random number between 0 and 99 for each packet in the secondary queue, and if this number is smaller than or equal to the loss probability-1 for the current link the packet is traversing, then the packet is lost.

Following is a pseudo code overview of the random packet loss implementation:

```
if (random_packet_loss_simulation_is_enabled){

  if (bursty)
    increment_or_reset_window_counter();

  for (each_packet_in_secondary_queue) {

    if (not_to_or_from_tap0) {
      rnd_num = generate_random_number_between_0_and_99();

      if ((NOT bursty &&
           ((linkstatus(src,dst) == 2 && rnd_num <= RNDCONST_LINK2-1) ||
            (linkstatus(src,dst) == 1 && rnd_num <= RNDCONST_LINK1-1)
           )
          )
         ||
          (bursty && (window_counter <= RANDSIMU_BURSTY_HI) &&
           ((linkstatus(src,dst) == 2 && rnd_num <= RNDBURST_HI_LINK2-1) ||
            (linkstatus(src,dst) == 1 && rnd_num <= RNDBURST_HI_LINK1-1)
           )
          )
         ||
          (bursty && (window_counter >= RANDSIMU_BURSTY_HI+1) &&
           ((linkstatus(src,dst) == 2 && rnd_num <= RNDBURST_LO_LINK2-1) ||
            (linkstatus(src,dst) == 1 && rnd_num <= RNDBURST_LO_LINK1-1)
           )
          )
         ) {
        remove_non_existent_packets_after_lost_packet();
        remove_lost_packet();
      }
    }
```

```
  }
}
```

In order to be able to switch between high and low bursty packet loss, a window counter was introduced. This counter is incremented each packet window, and when it is larger than `RANDSIMU_BURSTY_HI`, the burst is low. When the counter reaches its maximum value, `RANDSIMU_BURSTY_LO`, it is reset to start from 1, and the burst is high. This introduces another issue that should be considered: How often should we enable and disable bursty random packet loss? Currently, it is enabled between windows 1 and 50, and disabled between 51 and 100. On the test machine using test scenario 1 this means that bursty random packet loss is enabled and disabled about every 5 seconds. [1] showed delivery probabilities for bursty links to be much more varying and unpredictable than this, but there is really not a correct or wrong answer to how often the bursty loss should be enabled and disabled to make it realistic. It all depends upon each individual link and its network conditions at any given time. The main goal of implementing a bursty alternative to the constant one, was to obtain the same amount of packet loss on both options, only in two different ways. This means that an alternative solution where the enabling and disabling of the burst is more random or unpredictable, which might be more realistic, would give us less control over the total packet loss. Thus, the total in a bursty experiment would not necessarily be in the same range as an constant loss experiment with the same loss percentages. A long period with or without bursty loss also seems reasonable, as it allows the effects of sudden interference and packet loss to be more pronounced on the nodes.

To determine how many packets to drop randomly, we used the exponential model used in NE [14], previously described in Sect. 3.3. It describes the FER as $K(d - D_0)^\beta + P_0$, which gives us a continuously increasing value. Since our two distance steps are discrete, we decided to use the middle of our distance steps as the distance d. Since we used 100 meters as D_0, the model starts at 100 meters. Thus, d was set to 175 for 100-250 meters, and 375 meters for 250-500 meters. For P_0 and K, we used the same values as the [14] example for an "open outdoor area". For β we used 2.7, as this value seems to be closest to the example graph in Fig. 3.1, as well as providing reasonable values. Using 2 as β gives a FER of only 0.0321 at 500 meters, that is 3%, which seems too low to be realistic. Using 3 as β, like in their example, is too high for communication at 500 meters. Even 2.7 has a FER of more than 1.0 at 500 meters, but at 375 meters 2.7 gives a FER of 0.7714, which seems reasonable. Furthermore, 2.7 gives us a FER of 0.0232 at 175 meters, which fits well with their constant loss model of 2-3%. With this, we chose to use 3% random packet loss on links of 0-250 meters, and 75% on the 250-500 meter links. A further literature study should probably be done to see whether other FER models exist. This could be interesting if it were combined with an investigation into how many distance steps NEMAN could support without loosing

too much performance. I.e., the investigation should combine different FER models with a more continuous distance value than our current two step model.

For the gray zone implementation, a high FER in the gray zone range is very important. Without significant packet loss in the gray zone, the routing daemons will discover most of their routes over gray zones. These routes are then useless, thus inhibiting the majority of the multi-hop communication in the emulator. Routes over gray zones often seem to be the shortest and most preferable ones, as they cover the longest distances over fewest hops. A routing daemon that favors routes over as few hops as possible, might therefore produce mostly useless routes if the FER is too low in the gray zone. The natural alternative is to make the gray zone smaller, e.g. so that it only covers the distance between the maximum transmit distances of 2 Mbps and 1 Mbps in IEEE 802.11b. This way the amount of routes with gray zone hops would be smaller. However, without any proper data on the transmit range of the different speeds of IEEE 802.11b, as well as no link speed simulation in NEMAN, this solution is sufficient for now, as it gives us the effect of gray zones, as well as allowing communication.

It is important to note that the values we have discussed here are only recommendations, and by no means the "correct ones". How much loss it would be in a real MANET like this, can only be found by actually developing it and running measurements. The actual FER varies from network to network, and link to link. It all depends on the network conditions of the MANET at any given time, which means there is really no correct answer, only good and bad models for the average FER. With this in mind, we emphasized easy configuration of the loss-percentage, so that it can be tailored to fit each target network and experiment.

An alternative for the bursty mode, would be to turn the packet loss completely on and off, which was the original plan, instead of switching between 50% and 100%, as we do now. This means that we would have to lose 2*75% when the burst is on, which is not possible. Thus, we can not achieve the same total loss with the bursty mode as with the constant mode, which was the main goal of the option, when the packet loss is larger than 50%. It is also worth considering that it might not be very realistic with a link that is perfect half of the time, and almost useless the rest of the time, especially when the distance is large.

4.5 Discussion and experiments with the PCW

Having a large number of nodes sending traffic on a large number of TAP interfaces at the same time, the Topology Manager gets a long queue of packets from the Linux kernel. If the PCW is set too low, two packets that would have collided if they were alone in the emulated network, might not do so if other traffic on other links and nodes

in the same emulated network get in between them in the queue. Even the emulation of a large network in itself might prove to give enough load on a slower CPU so that the time between the processing of the packets would get larger than it would on a more powerful computer, resulting in no collision. On the other hand, in a small emulated network with little traffic, the amount of collisions might get unrealistic high if the PCW is too large. Thus we have made the PCW value configurable, so that it can be set to match the size of the emulated network, the amount of traffic the user of the emulator is expecting to introduce, and the speed of the computer that is used for the Topology Manager.

We now investigate three different emulation scenarios and try to determine which PCW values can be used with the different scenarios. For the experiments the machine running the Topology Manager and routing daemons is an Intel Pentium 4 2.4GHz with 512MB RAM, running Linux 2.4.21. The machine running the GUI is an Intel Pentium 4 3GHz with 1GB RAM, running Linux 2.6.9. The two machines are connected through a LAN. All the results here only apply when this setup is used for NEMAN, but the discussion will still give a good general indication of what PCWs that are useful, and how the PCW might be determined through similar experiments in other setups. Furthermore, traffic collision simulation was the only simulation scheme enabled during the experiments. For the experiments here we use 1/10th of the PCW as `SLEEP_TIME`, as this seems to be a good choice. We get back to the `SLEEP_TIME` discussion in Sect. 4.6.

A starting point for obtaining an value for the PCW is by measuring how much time the main thread of the Topology Manager uses to process each packet, Packet Processing Time (PPT), as this is the lowest value we are able to measure between the arrival of each packet. The time it takes to process an incoming packet and add it to the main queue depends on the number of hops the packet is supposed to traverse and the number of nodes in radio range on each of the hops. By getting some values for this time, we can get an idea of what the lowest useful PCW might be. Measuring the PPT indicates that the main thread typically uses about 30μs for data packets from nodes with no other nodes in receiving range. This includes receiving a packet from the Linux kernel and adding it to the main queue. A packet with 4 receivers in direct range typically has a PPT of 100μs. A packet sent over 6 hops in a 10 node scenario took about 300μs to process. This indicates that the lowest useful PCW in a 10 node scenario is around 1000μs, as this should allow a couple of multi-hop packets to be processed within the same window.

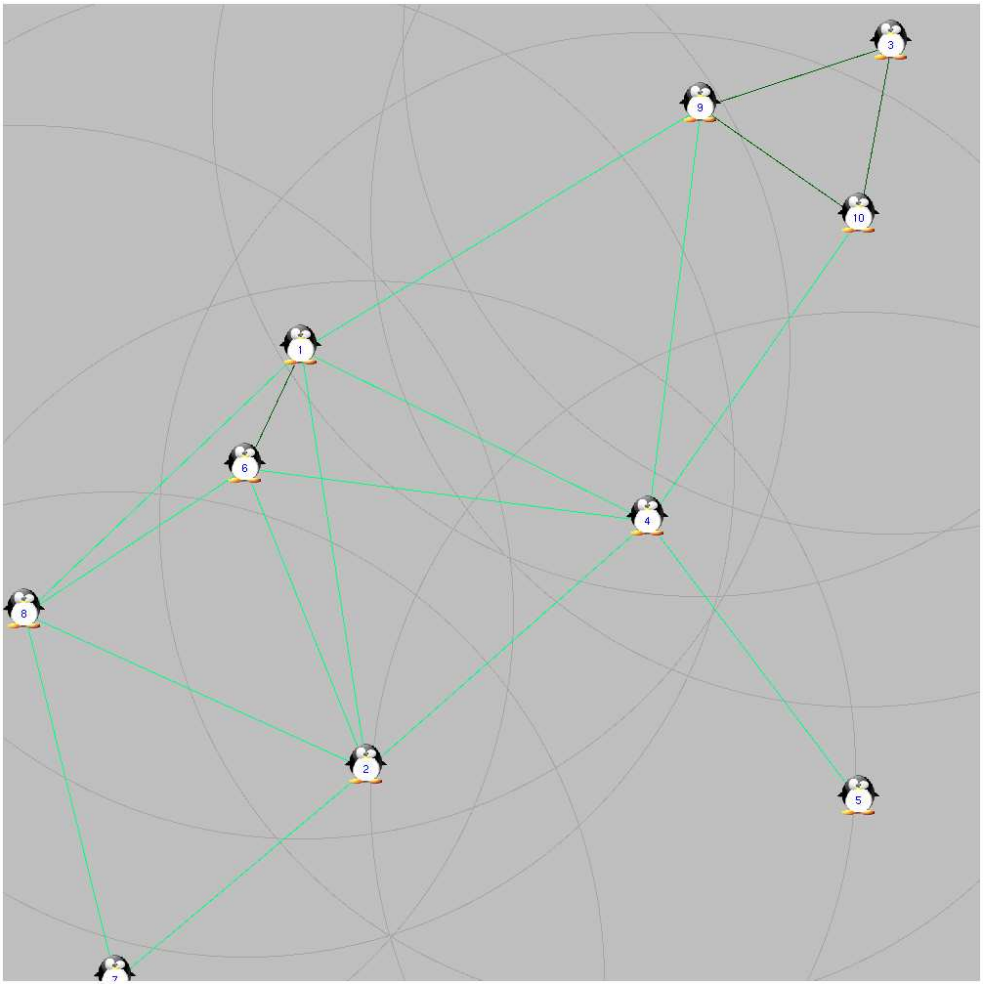

Figure 4.8: Screenshot from the NEMAN GUI at the start of test scenario 1

4.5.1 Test scenario 1

Test scenario 1: *10 nodes in a 1000x1000 meters area, the nodes are spread well over the area, little clustering, high speed (max 3), the only traffic is generated by the routing daemons, scenario is looping when needed. See Fig. 4.8 for a screenshot of the scenario at time 0.*

Test runs with test scenario 1 are done, using different PCW values, to see how many collisions are generated between the OLSR daemon broadcast messages on each step. Each run is stopped when the traffic collision simulation is as close as we can get to 5000 packets, after which the number of lost packets is rounded off to the closest ten. We get an indication of the possible range for the PCW by looking at the test results in Table 4.1. In Fig. 4.9 a plot of the results can be found as well.

As we see, a PCW of 1 000 000μs is clearly too large, as nearly all the packets got lost. We also see that a PCW of 100μs is too small for collisions to occur, which verifies the previous measurement where the main thread used about this much time to process one packet with four receivers. The reason for the collisions in one of the 100μs test runs seems to be that one of the nodes involved in the collision did the same trans-

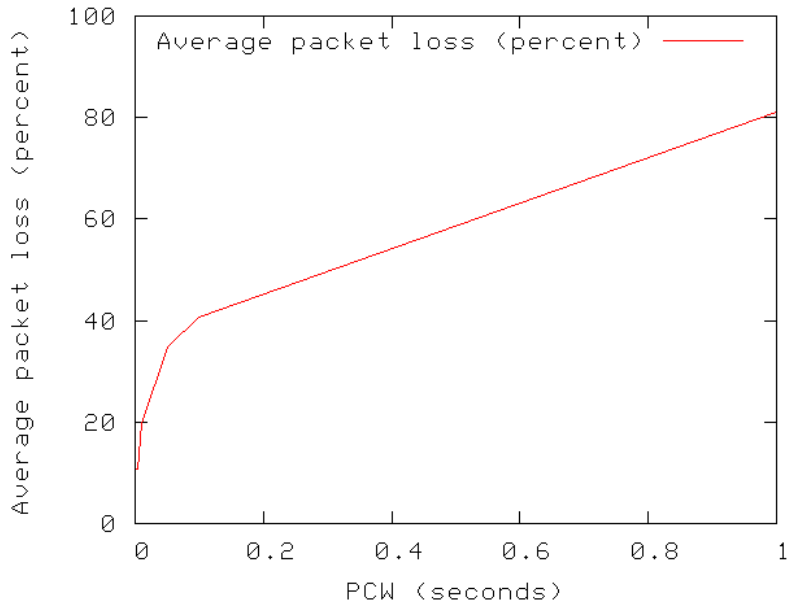

Figure 4.9: A plot of the results from the PCW experiments on test scenario 1

PCW	SLEEP_TIME	Packets collided	Percent	Average
1 000 000 μs	100 000 μs	4000 of 5000	80.0%	
		4070 of 5000	81.4%	81.1%
		4090 of 5000	81.8%	
100 000 μs	10 000 μs	2020 of 5000	40.4%	
		2010 of 5000	40.2%	40.6%
		2060 of 5000	41.2%	
50 000 μs	5000 μs	1800 of 5000	36.0%	
		1800 of 5000	36.0%	34.9%
		1630 of 5000	32.6%	
10 000 μs	1000 μs	1040 of 5000	20.8%	
		1020 of 5000	20.4%	19.6%
		880 of 5000	17.6%	
5000 μs	500 μs	690 of 5000	13.8%	
		550 of 5000	11.0%	10.8%
		380 of 5000	7.6%	
1000 μs	100 μs	670 of 5000	13.4%	
		540 of 5000	10.8%	10.9%
		430 of 5000	8.6%	
500 μs	50 μs	560 of 5000	11.2%	
		610 of 5000	12.2%	9.7%
		290 of 5000	5.8%	
100 μs	10 μs	0 of 5000	0.0%	
		0 of 5000	0.0%	0.1%
		10 of 5000	0.2%	

Table 4.1: Results from PCW experiments on test scenario 1

mission twice back to back, possibly indicating that using a PCW as low as this made NEMAN unable to process the packets from the Linux kernel fast enough, and thus making them queue up. We also observe that the 5000μs, 1000μs and 500μs tests had overlapping results and the average packet losses in the three tests were almost the same. The 1000μs tests even had slightly more loss than the 5000μs tests, showing that the results of each emulation run are very varying. This also indicates that a PCW value in the range between 5000μs and 500μs might be what we are looking for. Going lower than 500μs should not be done, as 500μs might already be too low for a collision between two multi-hop packets to occur, as discussed earlier. Even though 500μs works fine for the routing daemon broadcast collisions, it is probably too low if we want to have two multi-hop data packets traversing the network within the same window.

4.5.2 Test scenario 2

Test scenario 2: *10 nodes in a 800x800 meters area, the nodes are close, many links/high clustering, low speed (max 1), the only traffic is generated by the routing daemons, scenario is looping when needed. See Fig. 4.10 for a screenshot of the scenario at time 0.*

We now repeat the same experiment with test scenario 2. We expect to see more traffic collisions in this scenario as there are more links between the nodes than in test scenario 1. The results are found in Table 4.2 and Fig. 4.11 shows a plot of them.

The results confirm our expectations as the number of collisions in this scenario is larger than in test scenario 1. For most of the PCW values we have about 10% more packet loss due to collisions. We also see that the 5000μs, 1000μs and 500μs tests have results in the same range here as well. 500μs even got a few percent more than 5000μs, which would indicate that events in the emulation that are out of our control have a larger effect on the level of packet loss than the PCW. These events would probably be the scheduling between the different routing daemons and NEMAN itself, and possibly network delays between the computer running the GUI and the one running the Topology Manager. Furthermore, the gap between 10 000μs and 5000μs is smaller than in the previous scenario. This indicates that a PCW value in the range between 10 000μs and 500μs might be good for a scenario such as this.

4.5.3 Test scenario 3

Test scenario 3: *100 nodes in a 1000x1000 meters area, the nodes are very close, very many links/high clustering, low speed (max 1), the only traffic is generated by the routing daemons, does not need to loop as the GUI is too slow for it to complete. See Fig. 4.12 for a screenshot of the scenario at time 0.*

In this scenario, large amount of nodes and the huge amount of links means we are expecting packet loss to be very high. We have to do some adjustments to our experi-

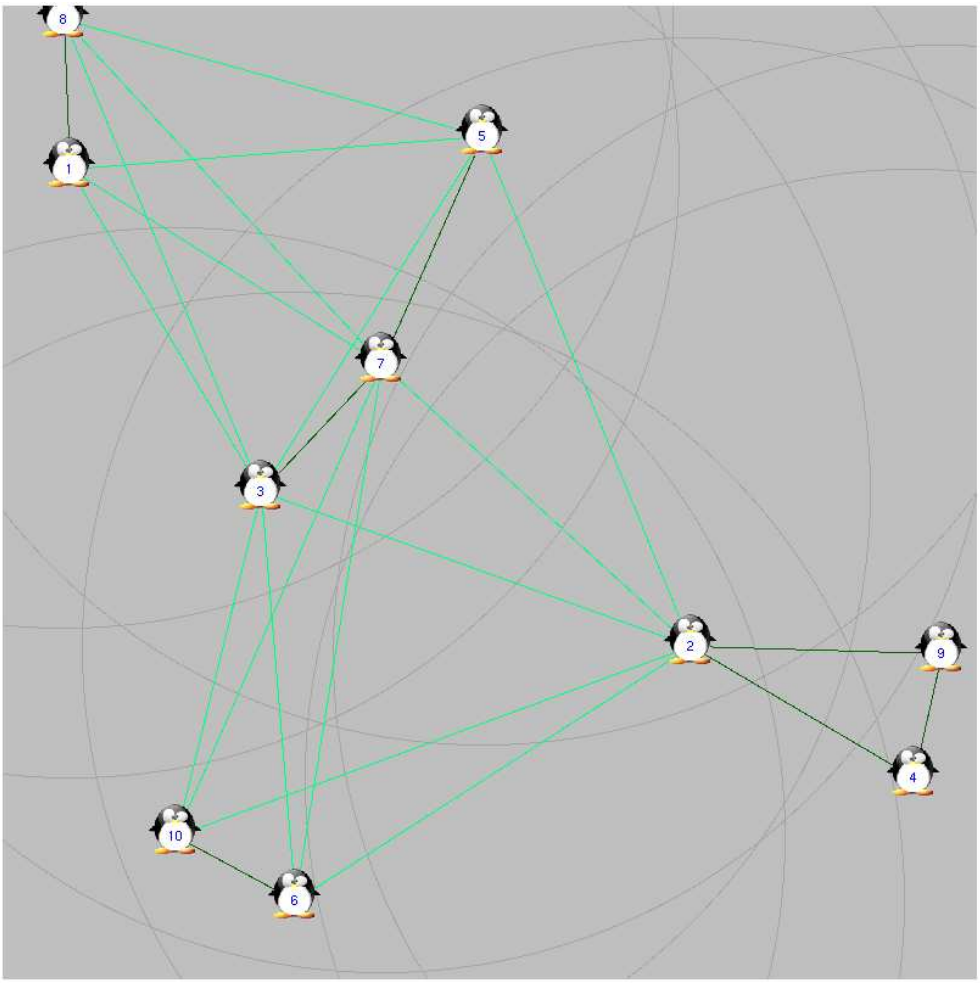

Figure 4.10: Screenshot from the NEMAN GUI at the start of test scenario 2

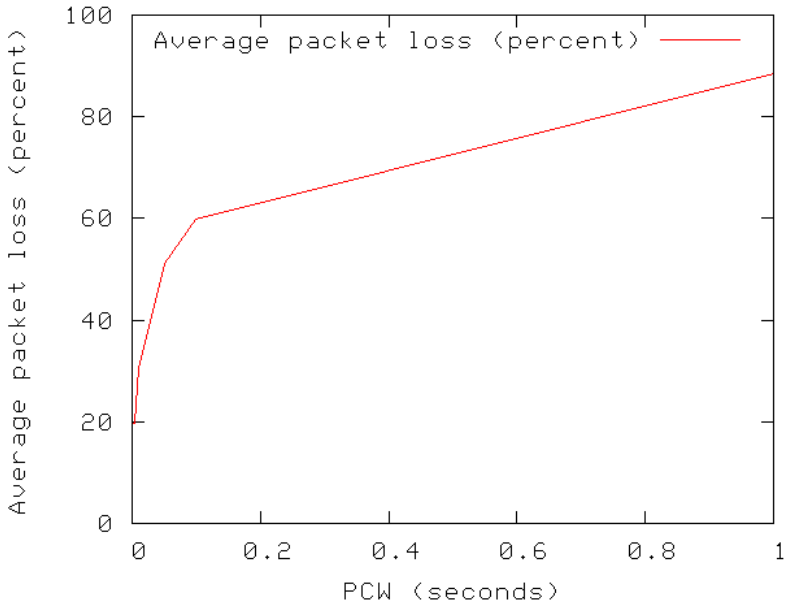

Figure 4.11: A plot of the results from the PCW experiments on test scenario 2

PCW	SLEEP_TIME	Packets collided	Percent	Average
1 000 000 μs	100 000 μs	4420 of 5000	88.4%	
		4420 of 5000	88.4%	88.4%
		4420 of 5000	88.4%	
100 000 μs	10 000 μs	3030 of 5000	60.6%	
		3050 of 5000	61.0%	59.8%
		2890 of 5000	57.8%	
50 000 μs	5000 μs	2570 of 5000	51.4%	
		2540 of 5000	50.8%	51.2%
		2570 of 5000	51.4%	
10 000 μs	1000 μs	1640 of 5000	32.8%	
		1460 of 5000	29.2%	30.8%
		1520 of 5000	30.4%	
5000 μs	500 μs	1200 of 5000	24.0%	
		880 of 5000	17.6%	19.8%
		890 of 5000	17.8%	
1000 μs	100 μs	700 of 5000	14.0%	
		1120 of 5000	22.4%	19.9%
		1160 of 5000	23.2%	
500 μs	50 μs	1100 of 5000	22.0%	
		1290 of 5000	25.8%	21.5%
		840 of 5000	16.8%	
100 μs	10 μs	10 of 5000	0.2%	
		0 of 5000	0.0%	0.1%
		0 of 5000	0.0%	

Table 4.2: Results from PCW experiments on test scenario 2

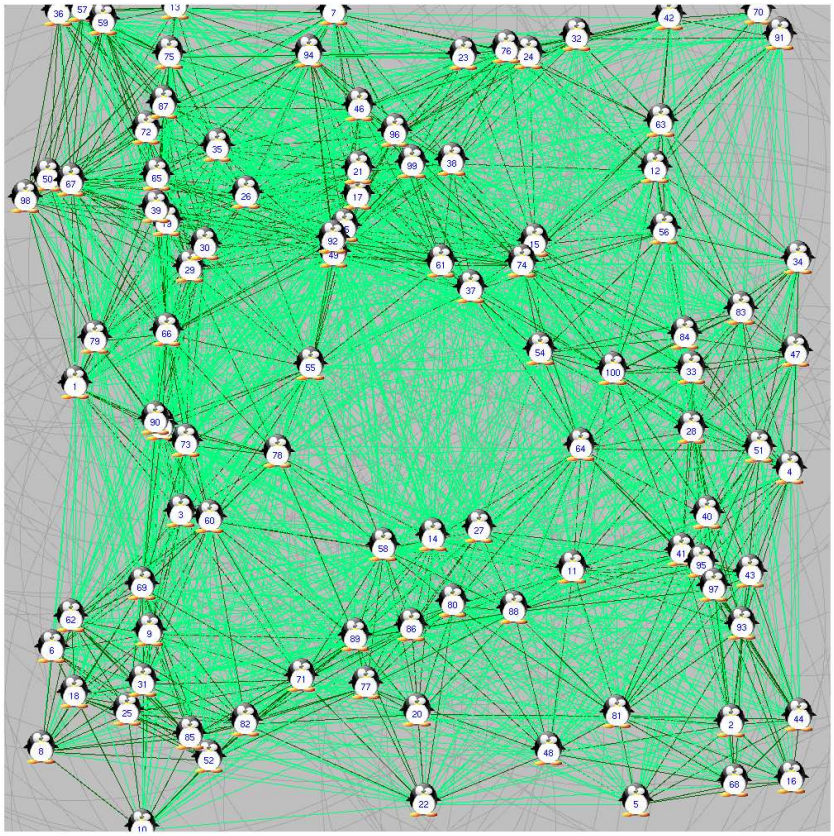

Figure 4.12: Screenshot from the NEMAN GUI at the start of test scenario 3

ment to make this work. First, we have to remove most of the log-file output from the Topology Manager. If we do not do this, it ends up spending too much of the processing time on writing to the log-file because of the large number of events. The queue of packets from the kernel then grows faster than the Topology Manager manages to process them. The log-file also grows too quickly so that it fills up the hard drive and tailing it over the network is not possible. We only need the collision simulation summary for this experiment, so this is the only log-file output that we keep. To be able to easily switch between different levels of log-output, a constant named `VERBOSE` was added. `VERBOSE` currently supports four different levels of information, where a higher level includes all the lower levels:

- **0** - Only NEMAN start/restart/stop and error information. This level is used as default when the log-file is not monitored.
- **1** - Summary information for each window and each simulation scheme, as well as UDP link reset/enable/disable commands.
- **2** - Details on each simulation scheme, and UDP route commands.
- **3** - Output from the queue-functions, and additional debug information.

PCW	SLEEP_TIME	Packets collided	Percent	Average
1 000 000 μs	100 000 μs	309 000 of 316 000	97.7%	
		300 000 of 307 000	97.7%	97.7%
		329 000 of 337 000	97.6%	
100 000 μs	10 000 μs	295 000 of 306 000	96.4%	
		295 000 of 305 000	96.7%	96.6%
		294 000 of 304 000	96.7%	
50 000 μs	5000 μs	279 000 of 307 000	90.9%	
		279 000 of 307 000	90.9%	91.1%
		278 000 of 304 000	91.4%	
10 000 μs	1000 μs	195 000 of 306 000	63.7%	
		199 000 of 313 000	63.6%	63.4%
		193 000 of 307 000	62.9%	
5000 μs	500 μs	128 000 of 310 000	41.3%	
		134 000 of 306 000	43.8%	42.8%
		132 000 of 305 000	43.3%	
1000 μs	100 μs	65 000 of 211 000	30.8%	
		61 000 of 211 000	28.9%	30.1%
		63 000 of 205 000	30.7%	

Table 4.3: Results from PCW experiments on test scenario 3

This means that `VERBOSE` is set to 1 for large experiments like this, and 2 for smaller experiments where the details are interesting. Secondly, 5000 packets are sent very fast, so we increase our number to something around 300 000, rounding the results off to the closest thousand. The results are found in Table 4.3 and Fig. 4.13 shows a plot of them.

The GUI is slow in terms of real time versus emulated time when emulating many nodes, compared to its speed when emulating only a few of them. Using 1 000 000μs as PCW, the emulated time/GUI clock runs for about 40 seconds before 300 000 packets are generated within this scenario, while it runs for about 880 seconds for the generation of 5000 packets in test scenario 2. Since the nodes in both of these scenarios have the same low speed, test scenario 3 appears much more static than test scenario 2, as it does not change much during the time needed to reach 300 000 generated packets. This, combined with the larger number of packets, is probably why the variation between the results in each of these tests is much smaller than with the previous scenarios. More variation in the links between the nodes gives more events that can lead to different collision simulation outcomes, depending on which routing daemon does which broadcast when, as well as on the cumulative effects of the events.

Using 1 000 000μs as PCW, we observe that the Topology Manager does not manage to keep up with the flow of packets, as it wastes too much time waiting for the window to pass. When the OLSR daemons are stopped after about 300 000 packets

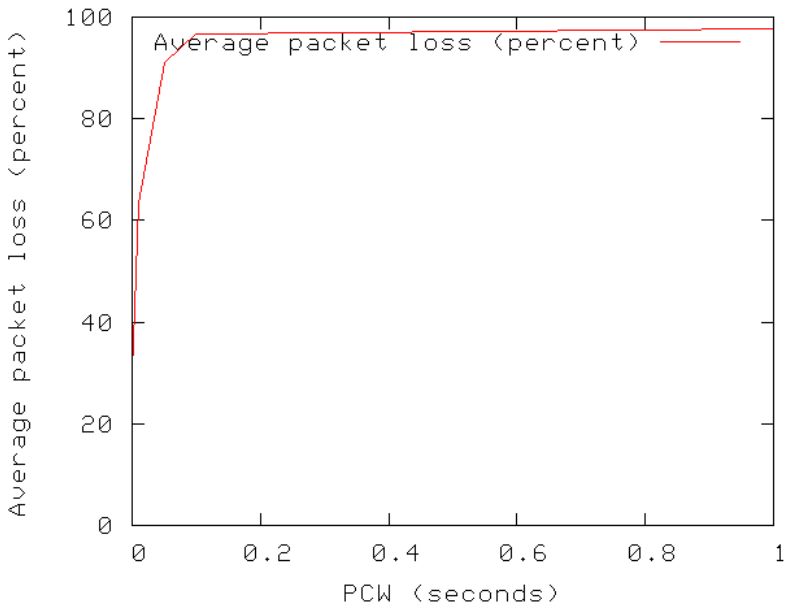

Figure 4.13: A plot of the results from the PCW experiments on test scenario 3

have been generated, it has about 100 000 packets left to process for collision simulation. It takes between 30 and 45 seconds to work through this queue. This problem is almost gone when we use 100 000μs as PCW, as it only has a queue taking between 1 and 2 seconds to process after we stop the OLSR daemons. When using 5000μs and below as PCW, the "opposite" happens; the window is too small to be able to work efficiently enough through the queue, and a large number of packets ends up in the queue awaiting processing. Using 5000μs it takes between 5 and 15 seconds to work through the queue after stopping the OLSR daemons. At around packet 280 000 when using 1000μs as PCW the Topology Manager had queued up enough packets to fill the entire main memory of the test machine (512MB). When this happened the emulation slowed down almost to a halt as the swap-space had to be used instead. This clearly showed that using this PCW and lower is unsuited when the number of nodes is that large. For the interest of the results themselves, the experiment was repeated using only 200 000 packets, but the 500μs and 100μs PCW tests were not done.

10 000μs seems to be close to optimal in terms of effective queue processing, so this PCW value was tested more thoroughly by running the test until 10 million packets had been generated. The Topology Manager managed to keep up with the generated packets and had under 1 second of packet processing left after the OLSR daemons were stopped when 10 million packets were reached. The test took about 26 minutes, and the GUI clock reached about 840s. The memory usage while running the emulation did not increase more than a couple of MB. The number of packets lost in collisions was about 8.7 millions, which gives us about 87% loss. This is significantly higher than the 63.4% in the previous 10 000μs tests, but this is to be expected due to the increased

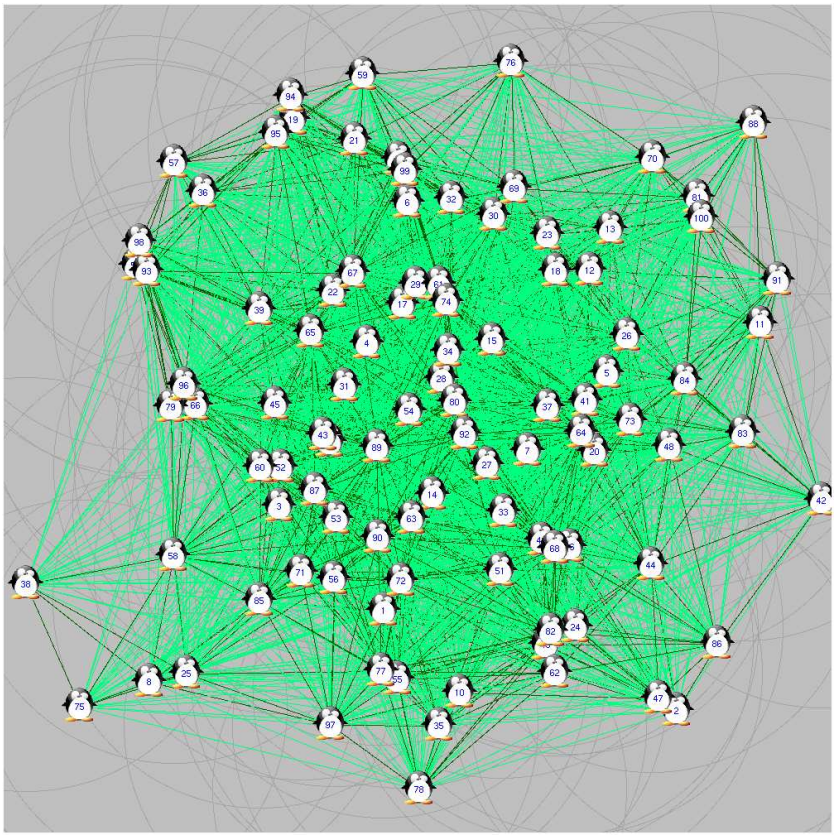

Figure 4.14: Screenshot from the NEMAN GUI at time 840s in test scenario 3

number of links in the scenario at time 840s. As previously discussed the scenario moves very slowly in the GUI when the amount of nodes is this large. The nodes in this scenario start off all over the area, and move toward the center, which means that the number of links is increasing as time moves along, giving us more and more loss due to collision. See Fig. 4.12 for a screenshot of the GUI at the start of test scenario 3, and Fig. 4.14 for the same scenario at time 840s.

4.5.4 Success rate for multi-hop packets

We now investigate the success rate when sending a data packet over multiple hops on two distinct scenarios, using three different values for the PCW. `SLEEP_TIME` is always set to 1/10th of the PCW. It is expected that the smaller the window is, the higher the success rate will be, as a larger window means more packet loss to the OLSR HELLO messages, giving fewer discovered routes. A larger window also increase the chance of collision between our data packet and OLSR traffic. The data packet is sent 20 times every 10 seconds, using the *send* utility distributed with the original NEMAN. The program generates an UDP packet on a given TAP interface, to a given IP address and a given port, which makes it a useful tool for doing single packet tests like this with NEMAN. The progress of the packet through the virtual network topology is

Test scenario	PCW	Succeeded	Failed	Routing Loop
1 at 100s	1000 μs	12	6	2
1 at 100s	5000 μs	10	10	0
1 at 100s	10 000 μs	8	12	0
1 at 100s	50 000 μs	0	20	0
2 at 50s	1000 μs	15	3	2
2 at 50s	5000 μs	14	6	0
2 at 50s	10 000 μs	5	14	1
2 at 50s	50 000 μs	0	20	0

Table 4.4: Results from PCW multi-hop packet success rate experiments on test scenario 1 and 2

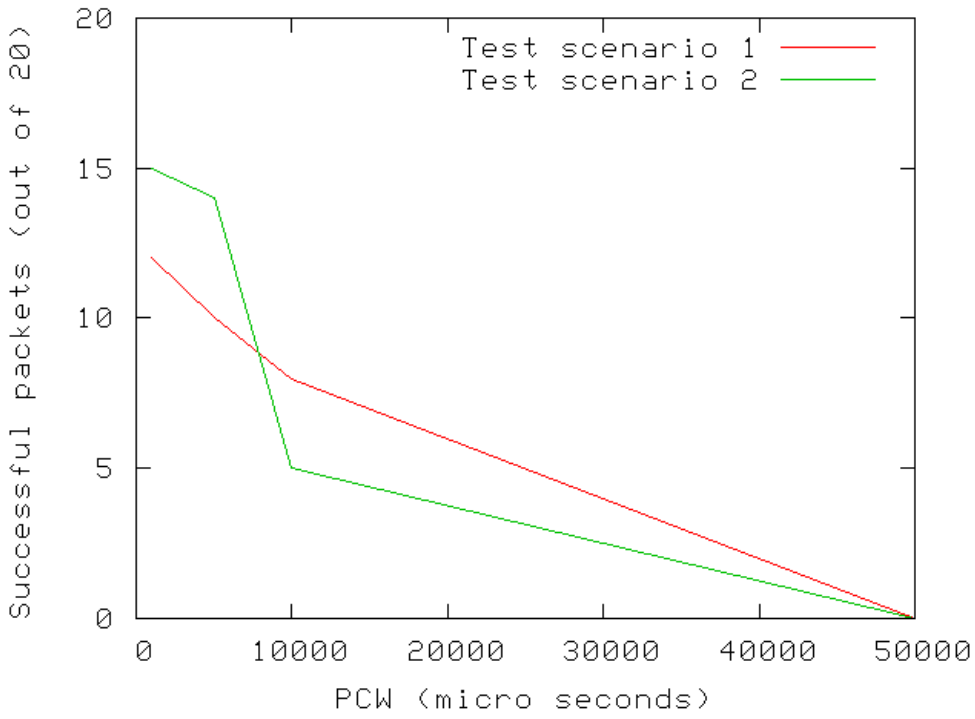

Figure 4.15: A plot of the results from the PCW multi-hop packet success rate experiments on test scenario 1 and 2

monitored by listening for packets on the tap0 interface using *tcpdump*. Test scenario 1 is used at scenario time 100s, where we send multi-hop packets from node 7 to node 3. The shortest route is 4 hops. Test scenario 2 is used at scenario time 50s, where we send multi-hop packets from node 8 to node 4. The shortest route is 3 hops. The results are found in Table 4.4 and Fig. 4.15 shows a plot of them.

As expected, the results show that the number of successful multi-hop packets decrease with increasing PCW value. The results for test scenario 2 are slightly better, and this is expected as our communication traversed 1 hop less in that scenario. Using 50 000μs gives us no successful multi-hop packets on either scenario, even though they

were both completely static. This clearly suggests that using a PCW as large as this is unsuited for any scenario. It is interesting that the difference in the average packet loss for 50 000μs and 10 000μs was only about 15%, (34.9% and 19.6% respectively), and yet this was enough to inhibit communication over 4 hops.

4.5.5 Conclusion for the PCW experiments

For smaller scenarios like test scenario 1 and 2, a PCW between 5000μs and 500μs gives us almost the same amount of packet loss due to collisions. This suggests that a PCW difference in this range does not affect the outcome of the collision detection. Remembering that it took NEMAN about 300μs to process a multi-hop packet sent over 6 hops in a 10 node scenario, it is probably best to use a PCW of 5000μs. This should allow a good amount of traffic to be processed, while at the same time not affecting the amount of packet collisions. We also saw that using 10 000μs as PCW works fine on both scenarios, but the amount of collisions is larger, as confirmed by the results from the multi-hop experiments in the previous section. Whether 5000μs or 10 000μs is to be used probably depends on the nature of the experiment, but keeping in mind that we have no collision avoidance techniques on our nodes, 5000μs is probably the best choice.

For a large scenario like test scenario 3, using a larger PCW than 10 000μs seems unsuited, especially if the goal of the experiment is to see some actual communication over multiple hops. Even 1000μs giving 30% packet loss is probably too high for establishment of longer routes, if we compare it to the results found on the multi-hop packet success experiments in the previous section. On the other hand, it is probably realistic with an average packet loss in the 90%-100% range when so many nodes are communicating over a smaller area with no collision avoidance techniques. The addition of collision avoidance in an experiment of this sort would most certainly reduce the number of collisions significantly. It might also be that 10 000μs is too small to allow NEMAN to process all the packets when each node has as many receivers as in this scenario. So for now a good PCW for scenarios such as test scenario 3 is probably in the range between 10 000μs and 50 000μs, depending on the amount of collisions the user expects. The amount of collisions with a PCW larger than 10 000μs might prove to be the most realistic as there is no collision avoidance, but it would probably hinder communication. 10 000μs might allow communication even without collision avoidance, but could be too small when many nodes wish to communicate at the same time. For future work it would be very interesting to see CSMA/CA implemented on the virtual nodes of NEMAN.

Another possibility for further work on NEMAN might be to make some kind of self test that automatically determines what the PCW should be for any setup. The

user could tell NEMAN what kind of experiment it is to be used for, and NEMAN could test the hardware and use this information together to choose a good PCW. The general rule is that the PCW would be increased with the number of nodes and the amount of traffic, while it would be decreased with the speed of the computer running the Topology Manager.

4.6 Discussion and experiments with SLEEP_TIME

The `SLEEP_TIME` value is used in the sub-thread to determine how long it should sleep while waiting for the PCW to pass, and how long it should sleep when waiting for packets to arrive in the queue. Generally, waiting too short means the sub-thread is busy waiting, using CPU time that could be used by the main thread to process incoming packets from the kernel, or possibly even other processes on the same computer. Waiting too long means the current window might have passed and the sub-thread might waste time on moving backwards in a long queue of packets to determine where the window should have started. This gives us ineffective queue processing, as time is wasted waiting for packets to arrive after the window has passed, and then even more time is wasted on moving the window start to the proper place. Thus the `SLEEP_TIME` should ideally be closely related to the PCW, such that the sub-thread does not sleep more than a fraction of the current window. We have used 1/10th of the PCW as `SLEEP_TIME` in the PCW-experiments in the previous section. Generally, we know we have a good `SLEEP_TIME` value if we can find a good compromise where the sub-thread sleeps as seldom as possible, while keeping `SLEEP_TIME` as low as possible.

While measuring how long the sub-thread actually sleeps when `usleep()` is called, we found that it usually took around 20 000μs from before `usleep()` was called, until the sub-thread was running again, with `SLEEP_TIME` set to 500μs. This indicates that the kernel does not manage to schedule the sub-thread again quickly enough for the `SLEEP_TIME` to really make any difference.

This is confirmed by investigating how often the sub-thread sleeps on different fractions of the PCW on different PCW values. We do this by looking at how many times the sub-thread sleeps for `SLEEP_TIME` μs while waiting for the window to pass, and while waiting for packets. This is measured for each time the collision simulation is run on test scenario 2. Test scenario 2 is used as there are too many events in test scenario 3 for any sleeping to be needed when the PCW is 10 000μs or smaller. Our results are found by counting each time the two `usleep()` calls are executed, and dividing this by the total number of collision simulations. As the sub-thread "wait for packets sleep count" grows very quickly when there are no packets in the queue between the restart of the Topology Manager and the start of the experiment, all the counters are set to be reset when the "node enable" command is received by the main thread. The PCW val-

PCW	SLEEP_TIME	PCW-cnt/Col-cnt	PKT-cnt/Col-cnt
10 000 μs	500 μs	0.32	2.86
	1000 μs	0.32	3.17
	2000 μs	0.35	3.89
	5000 μs	0.40	3.95
5000 μs	250 μs	0.00	2.74
	500 μs	0.00	2.83
	1000 μs	0.00	3.06
	2500 μs	0.00	2.73
1000 μs	50 μs	0.00	2.72
	100 μs	0.00	2.63
	200 μs	0.00	2.51
	500 μs	0.00	2.95

Table 4.5: Results from `SLEEP_TIME` experiments on test scenario 2

ues we test are 10 000μs, 5000μs and 1000μs. The fractions of these used for `SLEEP_TIME` are 1/20, 1/10, 1/5 and 1/2. Each test is run until 1500 collision simulations have been completed, as this is enough for the results to be reasonably stable.

Table 4.5 summarizes the findings. The third column, PCW-cnt/Col-cnt, shows the average number of times the sub-thread slept while waiting for the window to pass, that is, the number of sleeps divided by the total number of collision simulation windows. The fourth column, PKT-cnt/Col-cnt, shows the average number of times the sub-thread slept while waiting for packets to arrive into the queue.

As we see, the results are somewhat counter-intuitive, as one would expect that the higher the `SLEEP_TIME` was, the lower the number of times the sub-thread slept would be, but it was not always the case. This confirms that the differences in our `SLEEP_TIME` values in this test does not affect how often the sub-thread actually sleeps at the two `usleep()` calls, since the actual sleep-time is much larger. Thus the differences from test to test here are most likely the result of random variations in each test run. We also note that the number of sleeps while waiting for packets was low at PCW 10 000μs, and almost none at 5000μs (only a few were observed). With this we can conclude that any fraction of the PCW is likely to be good enough as `SLEEP_TIME`. Setting the `SLEEP_TIME` higher than the PCW does not make sense, even though we know that the sub-thread most likely ends up waiting longer anyway.

Sleeping for about 20 000μs while waiting for the window to pass should not be a problem in practice, since NEMAN moves the window back when necessary. The delay from this sleep probably means that the processing of packets is delayed, but delayed processing is only a problem if the scenario is large. In a large scenario this sleep is unlikely to occur in the first place, due to the large number of events and traffic. There is really no good alternative, as the sub-thread can not move forward

with a window unless the PCW has passed. Making the sub-thread do busy waiting instead of sleeping, and by that not yielding its processing, is not a good option. This would steal processing time from the main thread and the applications that are used with NEMAN on the same machine, and then hindering them from producing the traffic that the sub-thread is actually waiting for.

4.7 Summary

In this chapter, we have discussed the implementation of traffic collisions, communication gray zones, and random packet loss. We have also documented experiments with the PCW, and determined that 5000μs is a good window for traffic collision simulation on small scenarios, while 10 000μs is best for the larger scenarios. Finally, we have seen that any fraction of the PCW is acceptable for use as the `SLEEP_TIME` setting, as the sub-thread always ends up sleeping for about 20 000μs. In the next chapter, more thorough tests are performed for the functionality evaluation, and finally we investigate the performance of the new NEMAN version.

Chapter 5

Evaluation

The evaluation of the new NEMAN implementation is divided into two parts. In Sect. 5.1, the simulation schemes are tested for functionality, both by sending a single packet and by using FTP for file transfer between two virtual nodes. In Sect. 5.2, the performance of the new NEMAN is evaluated through comparison with the original NEMAN implementation.

5.1 Functionality

The first set of tests investigates the different simulation schemes separately and closely for functionality. This is done for collision detection in Sect. 5.1.1, gray zone simulation in Sect. 5.1.2, and random packet loss in Sect. 5.1.3. Sect. 5.1.4 then investigates FTP file transfer using different combinations of the simulation schemes. `VERBOSE` is set to 2 in all the experiments, so that all the simulation details are shown. The interesting parts of the output to the log-file are included where appropriate.

5.1.1 Collision detection functionality tests

Experiment 1: *Static test scenario 1 at scenario time 150 seconds, PCW is 5000us. We send two single-hop packets (packet sent directly from the sender to the receiver), from two different senders to the same receiver, using the send-program (previously described in Sect. 4.5.4). The goal of the experiment is to investigate the effect of the window size on collision detection.*

See Fig. 5.1 for a screenshot of the scenario. First we do both sends back to back. The sending nodes are 8 and 10, and the receiving node is 1. This should give a collision at node 1 (IP 10.0.0.1). The command is as follows: `send tap8 10.0.0.1 3333 foo && send tap10 10.0.0.1 3333 foo`.

This produces the following output to the log-file, and as we see, the "actual packets" with ID 2 (node 8 to 1) and ID 7 (node 10 to 1), collided:

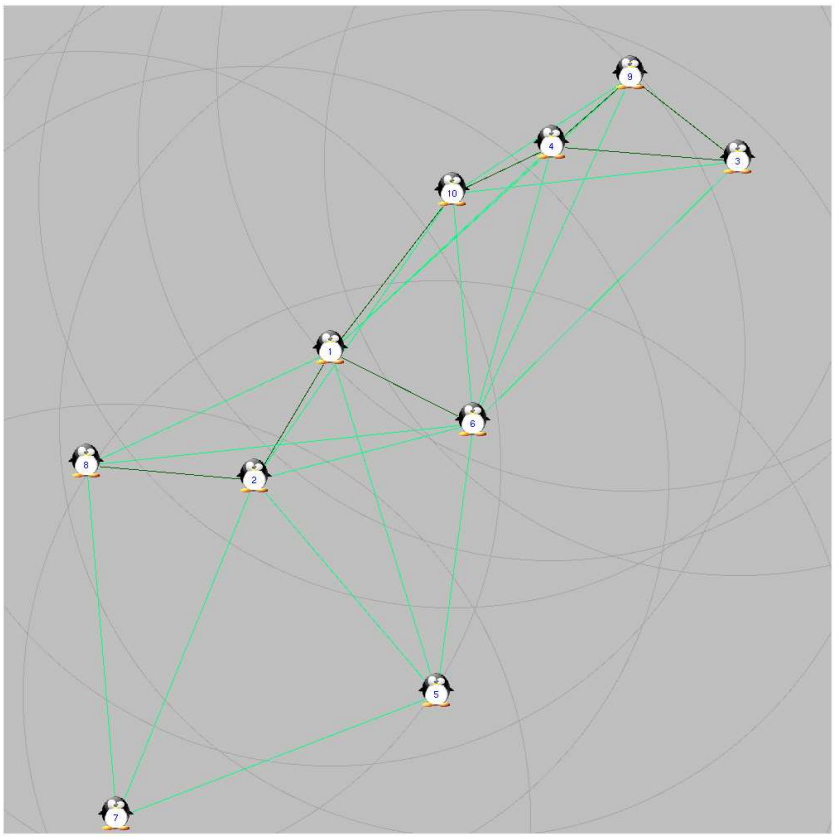

Figure 5.1: Screenshot from the NEMAN GUI at time 150s in test scenario 1

```
Packet ID 1/1, from 8 to 0, added to queue.
Packet ID 2/1, from 8 to 1, added to queue, actual packet.
Packet ID 3/1, from 8 to 2, added to queue.
Packet ID 4/1, from 8 to 6, added to queue.
Packet ID 5/1, from 8 to 7, added to queue.
Packet ID 6/2, from 10 to 0, added to queue.
Packet ID 7/2, from 10 to 1, added to queue, actual packet.
Packet ID 8/2, from 10 to 2, added to queue.
Packet ID 9/2, from 10 to 3, added to queue.
Packet ID 10/2, from 10 to 4, added to queue.
Packet ID 11/2, from 10 to 6, added to queue.
Packet ID 12/2, from 10 to 9, added to queue.
PCW 5000us has passed: window cur_time<->tail: 14159.917831us,
   window pcw_start<->tail: 1452.922821us.
->XX<- Collision between packets 2 and 7 occurred! ->XX<-
->XX<- Collision between packets 3 and 8 occurred! ->XX<-
->XX<- Collision between packets 4 and 11 occurred! ->XX<-
Collision simulation completed. 6 of 12 packets lost in collisions.
Packet ID 1/1, from 8 to 0, length 49, delivered.
```

```
Packet ID 5/1, from 8 to 7, length 49, delivered.
Packet ID 6/2, from 10 to 0, length 49, delivered.
Packet ID 9/2, from 10 to 3, length 49, delivered.
Packet ID 10/2, from 10 to 4, length 49, delivered.
Packet ID 12/2, from 10 to 9, length 49, delivered.
```

We also see that the packets 3, 8, 4 and 11 collided, but these are only radio-broadcast packets sent to other nodes in range. This can be seen by the second ID number, the hop-by-hop ID, which is the same for all the packets that originated from the same send-command. We also see the collision simulation summary, and finally the delivery of the packets that did not collide.

Ideally, we now would have wanted to do the same experiment, but with a *usleep* in between the two sends. The idea was to show that sleeping half the window would still result in a collision, while sleeping an entire window between the sends would divide them into two different windows, thus avoiding a collision. The command: `send tap8 10.0.0.1 3333 foo && usleep 2500 && send tap10 10.0.0.1 3333 foo`. This is unfortunately not possible, as a call to *usleep* in the shell results in a spacing of 10 000μs to 20 000μs between the two packet sends on the test machine, even if we only want to sleep for 1μs. This is the same problem as previously discussed in Sect. 4.6, where we saw that a sleep of 500μs while waiting for the PCW to pass, resulted in a wait of about 20 000μs before the sub-thread was running again. As we see in the log output below, the difference in arrival time between the two sends was 14 946μs. This is more than 5000μs, which means the window start is moved to the first packet and we get no collision:

```
Packet ID 1/1, from 8 to 0, added to queue.
Packet ID 2/1, from 8 to 1, added to queue, actual packet.
Packet ID 3/1, from 8 to 2, added to queue.
Packet ID 4/1, from 8 to 6, added to queue.
Packet ID 5/1, from 8 to 7, added to queue.
Sub-thread going to sleep while waiting for the PCW to pass.
Packet ID 6/2, from 10 to 0, added to queue.
Packet ID 7/2, from 10 to 1, added to queue, actual packet.
Packet ID 8/2, from 10 to 2, added to queue.
Packet ID 9/2, from 10 to 3, added to queue.
Packet ID 10/2, from 10 to 4, added to queue.
Packet ID 11/2, from 10 to 6, added to queue.
Packet ID 12/2, from 10 to 9, added to queue.
Sub-thread awoken, slept for 19995.000097us.
PCW 5000us has passed: window cur_time<->tail: 23313.045502us,
```

```
   window pcw_start<->tail: 14946.222305us.
Window start moved from ID 12/2 to 5/1, pcw_start<->tail now: 0.000000us.
Collision simulation completed. 0 of 12 packets lost in collisions.
Packet ID 1/1, from 8 to 0, length 49, delivered.
Packet ID 2/1, from 8 to 1, length 49, delivered.
Packet ID 3/1, from 8 to 2, length 49, delivered.
Packet ID 4/1, from 8 to 6, length 49, delivered.
Packet ID 5/1, from 8 to 7, length 49, delivered.
PCW 5000us has passed: window cur_time<->tail: 28360.843658us,
   window pcw_start<->tail: 0.000000us.
Collision simulation completed. 0 of 12 packets lost in collisions.
Packet ID 6/2, from 10 to 0, length 49, delivered.
Packet ID 7/2, from 10 to 1, length 49, delivered.
Packet ID 8/2, from 10 to 2, length 49, delivered.
Packet ID 9/2, from 10 to 3, length 49, delivered.
Packet ID 10/2, from 10 to 4, length 49, delivered.
Packet ID 11/2, from 10 to 6, length 49, delivered.
Packet ID 12/2, from 10 to 9, length 49, delivered.
```

In practice, this is not a big problem for NEMAN, as real applications will not depend upon calls to *sleep* or *usleep* in order to provoke or avoid collisions. Real traffic from other applications arrive at its "correct time" to NEMAN, and collides when it should.

Experiment 2: *Static test scenario 1 at scenario time 150 seconds, PCW is 5000us. We are sending one multi-hop packet from node 7 to node 9, and one single-hop packet from node 1 to node 6. The routing daemons are running. The goal of the experiment is to observe that the single-hop packet collides with the multi-hop packet, causing the rest of the multi-hop packets to become non existent.*

The scenario is the same as in experiment 1, see Fig. 5.1. We expect to see a collision between the multi-hop packet and the single-hop packet at node 8, 2 or 5, as the multi-hop packet from node 7 has to traverse one of these to reach node 9. Note that even though the actual data packet is sent from node 1 to 6, the radio transmission from this packet also reaches node 8, 2 and 5, where the collision with the multi-hop packet should occur. Seen from node 7, node 1 is a hidden node. The route is at least 3 hops, and reasonable routes might be as long as 5 hops. The sends are done back to back to ensure a collision between them. The command used was: `send tap7 10.0.0.9 3333 foo && send tap1 10.0.0.6 3333 foo`, resulting in the following log-output:

```
Packet ID 16250/2545, from 7 to 0, added to queue.
Packet ID 16251/2545, from 7 to 2, added to queue, actual packet.
```

```
Packet ID 16252/2545, from 7 to 5, added to queue.
Packet ID 16253/2545, from 7 to 8, added to queue.
Packet ID 16254/2545, from 2 to 0, added to queue.
Packet ID 16255/2545, from 2 to 1, added to queue, actual packet.
Packet ID 16256/2545, from 2 to 5, added to queue.
Packet ID 16257/2545, from 2 to 6, added to queue.
Packet ID 16258/2545, from 2 to 7, added to queue.
Packet ID 16259/2545, from 2 to 8, added to queue.
Packet ID 16260/2545, from 2 to 10, added to queue.
Packet ID 16261/2545, from 1 to 0, added to queue.
Packet ID 16262/2545, from 1 to 2, added to queue.
Packet ID 16263/2545, from 1 to 4, added to queue.
Packet ID 16264/2545, from 1 to 5, added to queue.
Packet ID 16265/2545, from 1 to 6, added to queue.
Packet ID 16266/2545, from 1 to 8, added to queue.
Packet ID 16267/2545, from 1 to 9, added to queue, actual packet.
Packet ID 16268/2545, from 1 to 10, added to queue.
Sub-thread going to sleep while waiting for the PCW to pass.
Packet ID 16269/2546, from 1 to 0, added to queue.
Packet ID 16270/2546, from 1 to 2, added to queue.
Packet ID 16271/2546, from 1 to 4, added to queue.
Packet ID 16272/2546, from 1 to 5, added to queue.
Packet ID 16273/2546, from 1 to 6, added to queue, actual packet.
Packet ID 16274/2546, from 1 to 8, added to queue.
Packet ID 16275/2546, from 1 to 9, added to queue.
Packet ID 16276/2546, from 1 to 10, added to queue.
Sub-thread awoken, slept for 19704.025239us.
PCW 5000us has passed: window cur_time<->tail: 21433.115005us,
   window pcw_start<->tail: 2156.019211us.
->XX<- Collision between packets 16251 and 16270 occurred! ->XX<-
Removing non-existent packet 16254 due to traffic collision.
Removing non-existent packet 16255 due to traffic collision.
Removing non-existent packet 16256 due to traffic collision.
Removing non-existent packet 16257 due to traffic collision.
Removing non-existent packet 16258 due to traffic collision.
Removing non-existent packet 16259 due to traffic collision.
Removing non-existent packet 16260 due to traffic collision.
Removing non-existent packet 16261 due to traffic collision.
Removing non-existent packet 16262 due to traffic collision.
```

```
Removing non-existent packet 16263 due to traffic collision.
Removing non-existent packet 16264 due to traffic collision.
Removing non-existent packet 16265 due to traffic collision.
Removing non-existent packet 16266 due to traffic collision.
Removing non-existent packet 16267 due to traffic collision.
Removing non-existent packet 16268 due to traffic collision.
->XX<- Collision between packets 16252 and 16272 occurred! ->XX<-
->XX<- Collision between packets 16253 and 16274 occurred! ->XX<-
Collision simulation completed. 3423 of 16254 packets lost in collisions.
Packet ID 16250/2545, from 7 to 0, length 49, delivered.
Packet ID 16269/2546, from 1 to 0, length 49, delivered.
Packet ID 16271/2546, from 1 to 4, length 49, delivered.
Packet ID 16273/2546, from 1 to 6, length 49, delivered.
Packet ID 16275/2546, from 1 to 9, length 49, delivered.
Packet ID 16276/2546, from 1 to 10, length 49, delivered.
```

As we see from the "actual packets" with hop-by-hop ID 2545, the route chosen for the multi-hop packet was 7->2->1->9, and it collided with the single-hop packet from node 1, which has hop-by-hop ID 2546, at node 2. The multi-hop packets that were supposed to be sent from nodes 2, 1 and 9 were correctly removed, as they are not supposed to exist once the packet collided at node 2. We also observe that the single-hop packet from node 1 collided with the multi-hop transmission at nodes 8 and 5 as well.

5.1.2 Gray zone functionality test

Experiment: *Static test scenario 1 at scenario time 150 seconds. We are sending one multi-hop packet from node 4 to node 2, via node 1. The link between node 4 and 1 is in the gray zone. The routing daemons are running. The goal of the experiment is to observe that a route is discovered between the two nodes, yet our multi-hop packet was lost over the gray zone hop, and the packets that were to be sent from node 1 to 2 are marked as non existent.*

The command was `send tap4 10.0.0.2 3333 foo`, which resulted in the following log-file output:

```
Packet ID 1380/217, from 4 to 0, added to queue.
Packet ID 1381/217, from 4 to 1, added to queue, actual packet.
Packet ID 1382/217, from 4 to 3, added to queue.
Packet ID 1383/217, from 4 to 6, added to queue.
Packet ID 1384/217, from 4 to 9, added to queue.
Packet ID 1385/217, from 4 to 10, added to queue.
```

```
Packet ID 1386/217, from 1 to 0, added to queue.
Packet ID 1387/217, from 1 to 2, added to queue, actual packet.
Packet ID 1388/217, from 1 to 4, added to queue.
Packet ID 1389/217, from 1 to 5, added to queue.
Packet ID 1390/217, from 1 to 6, added to queue.
Packet ID 1391/217, from 1 to 8, added to queue.
Packet ID 1392/217, from 1 to 9, added to queue.
Packet ID 1393/217, from 1 to 10, added to queue.
Removing non-existent packet 1386 due to gray zone.
Removing non-existent packet 1387 due to gray zone.
Removing non-existent packet 1388 due to gray zone.
Removing non-existent packet 1389 due to gray zone.
Removing non-existent packet 1390 due to gray zone.
Removing non-existent packet 1391 due to gray zone.
Removing non-existent packet 1392 due to gray zone.
Removing non-existent packet 1393 due to gray zone.
Removing packet 1381 due to gray zone.
Removing packet 1383 due to gray zone.
Gray zone simulation completed. 2 of 1385 packets lost in gray zones.
Packet ID 1380/217, from 4 to 0, length 49, delivered.
Packet ID 1382/217, from 4 to 3, length 49, delivered.
Packet ID 1384/217, from 4 to 9, length 49, delivered.
Packet ID 1385/217, from 4 to 10, length 49, delivered.
```

The experiment went as expected, and the radio-broadcast transmission from node 4 that was supposed to reach node 6, was lost over a gray zone as well. Also note that the gray zone packet loss counter does not include the non-existent packets.

5.1.3 Random packet loss functionality test

Experiment: *Static test scenario 1 at scenario time 150 seconds. The routing daemons are the only traffic generators. The experiments are done with both constant and bursty packet loss, two times each, for 5 minutes. Current time is used as seed for the random number generator, which means we have a different seed for each run. The goal of the experiment is to observe that routing daemon packets are lost randomly, and measure how high the actual loss is over time.*

Typical output to the log-file is as follows, where the simulation summary below is the result of the first constant loss run:

```
Packet ID 24510/3846, from 5 to 0, added to queue.
Packet ID 24511/3846, from 5 to 1, added to queue.
```

Mode	Run	0-250m	250-500m
Constant	1	2.72%	74.90%
	2	3.15%	74.79%
Bursty	1	2.80%	75.47%
	2	3.05%	74.84%

Table 5.1: Results from random packet loss functionality evaluation experiments

```
Packet ID 24512/3846, from 5 to 2, added to queue.
Packet ID 24513/3846, from 5 to 6, added to queue.
Packet ID 24514/3846, from 5 to 7, added to queue.
Removing packet 24512 due to random packet loss.
Removing packet 24513 due to random packet loss.
Random packet loss simulation completed:
     2.723343% (189 of 6940) lost on 0-250m links.
     74.901675% (10284 of 13730) lost on 250-500m links.
Packet ID 24510/3846, from 5 to 0, length 74, delivered.
Packet ID 24511/3846, from 5 to 1, length 74, delivered.
Packet ID 24514/3846, from 5 to 7, length 74, delivered.
```

As we see for this particular example, node 5 lost 50% of its packets (not counting tap0), which is expected, as all the links from node 5 are in the 250-500m range, (see Fig. 5.1). It is also worth noting that the total counters for the random packet loss simulation summary only include the packets that might be removed on each link type, and not tap0 packets or packets lost due to previously applied simulation schemes. The results of all four runs are summarized in Table 5.1.

All the results are within +/- 0.5% of the intended packet loss percentage, which is good. We also see that the total loss is the same on constant and bursty mode, which means the implementation works as it should.

5.1.4 FTP file transfer

We now investigate whether file transfer is possible over 2-3 hops on a static test scenario 1 at scenario time 150s, when combinations of the simulation schemes are used. A data file of 100KB is transferred between a FTP client and a FTP server, running on separate virtual nodes. If the 100KB experiments are successful, the same experiment is repeated with a 1MB file as well. The FTP client used for the experiment is a standard Linux *ftp* client, and the server is NcFTPd [17]. The goal of these experiments is to observe if any data gets through, if so how much, and if all data manages to get through, how much time it took. Combinations of the three simulation schemes are

Filesize	Run	Time	Avg. speed	Completed
100KB	1	0.24s	420KB/s	100%
100KB	2	0.239s	420KB/s	100%
100KB	3	0.239s	420KB/s	100%
1MB	1	1.44s	710KB/s	100%
1MB	2	1.36s	750KB/s	100%
1MB	3	1.98s	520KB/s	100%

Table 5.2: Results from FTP experiment 1 - No simulation schemes

used.

To enable test applications to run on a specified TAP interface, without rewriting them, the author of NEMAN wrote *libsocket.so*. This is a library which, when preloaded before an application, decides which net-device is to be used by *all* the sockets created in the application. NcFTPd is started on tap8 by using the following command: `SO_BINDTODEVICE=tap8 LD_LIBRARY_PATH=. LD_PRELOAD=libsocket.so ./ncftpd general.cf domain.cf`. Likewise, the FTP client is started on tap10, connecting to tap8/10.0.0.8: `SO_BINDTODEVICE=tap10 LD_LIBRARY_PATH=. LD_PRELOAD=libsocket.so ftp 10.0.0.8`.

As we see in Fig. 5.1, the shortest route between tap8 and tap10, is two hops, and reasonable routes might be established over three hops as well. Connecting the FTP client to the server is done before the experiments, and the number of retries is documented. If a data transmission is taking very long time due to high packet loss, it is aborted. The total data received by the client so far, the total time elapsed, and the average speed are then documented. *tcpdump* is used on tap0 to monitor communication when problems arise due to the simulation schemes.

Experiment 1: *No simulation schemes are enabled. This is done to get some results to compare to the other experiments. This experiment should give perfect results, similar to what one could expect with the original NEMAN version.*

The client connected successfully to the server on the first attempt, and as we see in Table 5.2, there were no problems with the transfers.

Experiment 2: *Traffic collision simulation enabled, the PCW is set to 5000µs. The transfer is expected to take significantly longer time than experiment 1 when collisions occur. 100KB should still be possible without too much delay if we are lucky and avoid collisions between our packets and the routing daemon packets.*

The client connected successfully to the server on the first attempt. The results are found in Table 5.3. During the first run, there were too many collisions, and the transfer died. The client gave no indication of problems during the 5 minutes, but lack of communication indicated that the server had timed out, and the transfer was not com-

Filesize	Run	Time	Avg. speed	Completed
100KB	1	330s	0.064KB/s	21%
100KB	2	0.881s	110KB/s	100%
100KB	3	0.48s	210KB/s	100%
1MB	1	55.6s	18KB/s	100%
1MB	2	156s	3.3KB/s	52%
1MB	3	238s	1KB/s	24%

Table 5.3: Results from FTP experiment 2 - Traffic collision simulation

Filesize	Run	Time	Avg. speed	Completed
100KB	1	0.399s	250KB/s	100%
100KB	2	0.68s	150KB/s	100%
100KB	3	1.76s	57KB/s	100%
1MB	1	39s	26KB/s	100%
1MB	2	20.2s	51KB/s	100%
1MB	3	44.8s	23KB/s	100%

Table 5.4: Results from FTP experiment 4 - Random packet loss

pleted. The next two runs had no problems, and as we see, they took slightly longer to complete than in experiment 1. This is probably due to the added collision simulation overhead and TCP retransmissions. One of the 1MB runs were also completed, with the rather low throughput of 18KB/s.

Experiment 3: *Gray zone simulation enabled. The transfer is expected to either work perfectly, or not work at all, depending on whether the route discovered between the nodes is over a gray zone or not.*

The client tried to connect to the server 10 times, but never got any response. All communication between the two nodes that appeared on tap0, was from the client to the server. This indicates that a route was discovered over a gray zone, which hinders a connection between the client and the server. If we look at Fig. 5.1, we see that all routes, except for 8->2->1->10, contain one or more gray zone links. Thus, this result is as expected, as there is no random packet loss over the gray zone links which could have hindered route discovery over these.

Experiment 4: *Random packet loss enabled. The speed of the transfer is expected to vary more or less like with experiment 2 since the actual loss is unpredictable, just like collisions between our FTP packets and the routing daemon packets.*

The client connected successfully to the server on the first attempt. As we see from the results in Table 5.4, all the 100KB and 1MB runs were completed successfully, but the actual speed was varying. Overall, the speed and success rate was better than the traffic collision runs in experiment 2. It might indicate that the good route via nodes 2

Filesize	Run	Time	Avg. speed	Completed
100KB	1	17.5s	5.7KB/s	100%
100KB	2	1.44s	69KB/s	100%
100KB	3	2.38s	42KB/s	100%
1MB	1	17.5s	58KB/s	100%
1MB	2	19.3s	53KB/s	100%
1MB	3	16.6s	62KB/s	100%

Table 5.5: Results from FTP experiment 5 - Gray zone simulation and random packet loss

Filesize	Run	Time	Avg. speed	Completed
100KB	1	230s	0.29KB/s	69%
100KB	2	423s	0.22KB/s	97%
100KB	3	115s	0.79KB/s	93%

Table 5.6: Results from FTP experiment 6 - All three simulation schemes enabled

and 1 (8->2->1->10) was chosen, as results this good are unlikely if the route contained 250-500 meter links with 75% loss.

Experiment 5: *Gray zone simulation and random packet loss enabled together. This is done to see whether the gray zone simulation works better when the packet loss is high on the 250-500 meter links. With some luck, we should manage to avoid the routes that have gray zone links, and the good route via nodes 2 and 1 should be used instead. The transfer is expected to either not work at all, or work as good as experiment 4, as this experiment with a non-gray zone route, is more or less the same as experiment 4.*

The client connected successfully to the server on the first attempt. The results are found in Table 5.5. All the runs completed with no problems, except the first 100KB run which took longer time than the others. This shows that 75% random packet loss on the 250-500 meter links is enough to mitigate OLSR route discovery over them, at least in our scenario where a better route is available. We know for sure that the 8->2->1->10 route was used, as this is the only route between node 8 and 10 with no gray zone links.

Experiment 6: *All three simulation schemes are enabled together. This is done to see if communication is possible, or if it is too limiting for a FTP session. It was hard to predict the outcome of this experiment, so we just started the experiment instead.*

The connect attempts failed ten times. Seven of these were with no answer from the server at all, and three of the attempts got a connection to the server, but failed after entering the username. This was probably because either the packet with the username, or the reply from the server asking for the password, got lost. Since this did not work, the same experiment was retried with the client on node 1 instead, thus

giving us two hops, instead of three. Now the client managed to get a connection on all the attempts, but it took longer time than with the previous experiments. As we see from the results in Table 5.6, all three file transfers failed before completion, but 97% and 93% of the file got through at the second and third attempts, respectively.

The results from experiment 6 show that using all three simulation schemes together, is not a good option if file transfers are done over the network. Communication is possible, and with a more "aggressive" protocol than FTP over TCP, it might still work good. FTP over TCP seems to time-out (i.e. "give up"), or use too long back-offs when the loss is high. It is hard to say if the results are realistic, but considering the lack of a MAC layer collision avoidance protocol, like CSMA/CA, the results would probably not have been better in a real MANET like this scenario.

5.2 Performance

In this section, we investigate the performance of the new version of NEMAN. Our goal is to investigate how many nodes the new version of NEMAN can emulate with routing daemons running and all simulation schemes enabled without getting behind on the processing, causing packets to queue up. In addition, the original NEMAN is run with the same experiments for comparison. We already saw in the PCW-experiments in Sect. 4.5 that the new NEMAN can handle 100 nodes and routing daemons on the test machine when 10 000μs is used for the PCW and only collision simulation is enabled. Are 100 nodes still possible when gray zone and random packet loss are enabled as well, and if so, how many more nodes are possible?

The testing is done by running experiments where the number of nodes are incremented by 25 for each experiment. All the scenario-files for these experiments are generated for these tests using the ns-2 *setdest* program with the following parameters: 1000x1000 meters, pause time 3, max speed 1, and scenario time 100 seconds. Each experiment is run for about 3 minutes. It is also worth noting that the scenario-files used with the original NEMAN are the original files generated by *setdest*, while the scenario-files used with the new NEMAN are processed by the *mkdist.pl*, previously discussed in Sect. 4.3. The experiments are done on the same test machines that were used for the PCW experiments, refer to Sect. 4.5 for their specifications.

In order to detect that NEMAN gets too slow on packet processing, these three methods are used:

- **Monitoring memory usage** - This can be done by using the standard Linux tool *top*, which displays memory and CPU usage for all the processes on the system. If the memory usage is rising very fast, we know that packets are queued up and NEMAN is too slow for this number of nodes.

# Nodes	PCW	Time	Mem use	Logfile delay	Send delay	Conclusion
100	10 000μs	3mins	Constant	None	None	Passed
125	10 000μs	3mins	Constant	None	None	Passed
150	10 000μs	3mins	Constant	None	None	Passed
175	10 000μs	3mins	Constant	None	None	Passed
200	10 000μs	30-60s	Rising	15s	3-4s	Failed
200	10 000μs	1-2mins	Rising	Too long	Too long	Failed
200	30 000μs	45s	Rising	Too long	Too long	Failed

Table 5.7: Results from the performance evaluation experiments with the new NEMAN

- **Monitor the log-file after stopping routing daemons** - If the processing continues for more than 1-2 seconds after the routing daemons are stopped, we know that packets were queued up.
- **Use the send utility** - Sending a packet with the *send* utility during the experiment, we can monitor tap0 using *tcpdump* to see how long it takes before the packet shows up on tap0, and thus determine the processing delay through NEMAN.

We start by investigating the new NEMAN version. The PCW is set to 10 000μs for all the experiments with the new NEMAN, as this value works good for large scenarios. A larger PCW should also be tried when the queue processing is too slow, in case the PCW is the cause of the problem. `VERBOSE` is set to 1 and random packet loss is done constantly. The experiments start at 100 nodes and the results are found in Table 5.7.

As we see, the tests with up to 175 nodes were completed without any problems. At 200 nodes the memory usage was rising so quickly that the routing daemons were stopped after less than 1 minute on the first run. After this it took about 15 seconds to work through all the packets that were left in the queue. A data packet was introduced about 15 seconds into the experiment, and it showed up on `tcpdump` after 3-4 seconds. The experiment was repeated, this time running between 1 and 2 minutes. A data packet that was introduced after about 45 seconds, had not shown up on `tcpdump` about 2 minutes after the routing daemons were stopped. Waiting longer than 2 minutes for the queue to be processed was not a good idea, as the memory usage did not appear to decrease, but rather the opposite. The experiment was repeated a third time, now with 30 000μs as PCW. The experiment ran for 45 seconds, and a *send*-command was issued after 30 seconds. After 2 minutes of waiting, the send packet had not shown up on `tcpdump`, and the Topology Manager had not completed the queue. This shows that increasing the PCW does not help, at least not enough to support 200 nodes on our test machine. Also, at 175 nodes or more, the GUI is getting very slow. At 200 nodes it took about 45 seconds to just start the experiment, and the load on the machine running the

#Nodes	Time	Mem use	Logfile delay	Send delay	Conclusion
175	3mins	Constant	None	None	Passed
200	3mins	Constant	None	None	Passed
225	3mins	Constant	None	None	Passed

Table 5.8: Results from the performance evaluation experiments with the original NEMAN

GUI was very high as long as the experiment was running.

As we know that the new NEMAN can handle 175 nodes, we can start our performance test with the original NEMAN at 175 nodes. We know that it is faster, as it has less processing to perform for each packet. Two changes were made to the original NEMAN in order to adjust it for this test: Time and date were added to the logfile-output and route information output was removed, while one output line for each packet to be sent was added instead. Doing this we are able to see if packets from the kernel are queued up when the routing daemons are stopped. The results are found in Table 5.8. From the results, we see the original NEMAN Topology Manager had no problems emulating 200 and 225 nodes. The GUI, on the other hand, performs badly at 175+ nodes, so testing anything higher than 225 nodes is unnecessary. Using a static scenario would avoid the GUI bottleneck, but NEMAN is meant to be used with mobility, and thus the performance results from a static scenario are not very interesting.

We have seen that the new NEMAN can handle scenarios of up to 175 nodes on our test machine when all three simulation schemes are enabled. The GUI was the biggest bottleneck with the original version of NEMAN. Considering that the test machine used for the GUI is significantly faster than the machine used for the Topology Manager, we see that the new NEMAN performs good enough to make them "equally large" bottlenecks instead. With this, we can conclude that the performance of the new NEMAN version is satisfactory, and that it does not limit the total number of nodes that NEMAN can reliably emulate when the current GUI is used. It is also important to point out that any test application running on some or all the nodes in addition to the routing daemons, might also use processing power. This might limit the total number of nodes in a large scenario, compared to these experiments where no additional applications where used with NEMAN.

5.3 Summary

In this chapter we have shown how the different simulation schemes work in practice, and seen that the implementation works as it should. We have also found the performance of the new NEMAN Topology Manager to be satisfactory, as its performance is good enough to avoid being a larger bottleneck than the GUI.

Chapter 6

Conclusion

6.1 Summary and concluding remarks

In this thesis, we have looked at some of the physical and MAC layer issues that affect MANETs. We have studied signal strength, traffic collisions, the hidden node problem, communication gray zones and random packet loss in detail in Chapter 2. All of these are issues that seriously affect wireless communication, and thus should be taken into account when developing an emulator for MANETs. The original NEMAN is a MANET emulator that forwarded packets based solely on node connectivity, and it lacked simulation of physical and MAC layer issues. During this master thesis the physical and MAC layer model of NEMAN has been improved to simulate the effects of some of these issues. NEMAN has been extended to simulate traffic collisions, communication gray zones, and random packet loss.

These three issues have the same outcome; they all cause packet loss. Thus, in order to be able to determine which packets that are lost, NEMAN was changed to queue up all generated traffic instead of just forwarding it as it arrives. As everything in NEMAN runs on the same computer, no two events are really at the same time. This presents a problem when we want to detect traffic collisions, since two or more packets that might have caused a collision, did not appear in the network at the same place at the same time. To meet this problem we had to define a collision window, which states how close in time two packets appearing at the same place in the emulated network have to be for a collision to occur. The size of this window is named Packet Collision Window (PCW), and extensive experiments with this value have been performed to determine the size of this window. The results indicate that good PCW values are 5000μs for small scenarios, and 10 000μs for large scenarios.

Communication gray zones are the radio ranges where only low speed data transmissions might make it to the receiver. In IEEE 802.11b, data packets are sent at high speeds, typically 11 Mbps, while broadcasts are usually done at 1 Mbps, thus provid-

ing a longer range than the data packets. For a routing daemon this means that routes discovered through broadcasts might not always be usable for data traffic. Wanting to simulate this, presented us with an additional problem: determining the size of the gray zone. Clearly, we should use the radio range of the different IEEE 802.11b transmit speeds. Obtaining general figures for these is not easy, as they differ on each network card, the transmit power used, the antenna, and the environment the card operates in. For simplicity, we decided that using only two transmit ranges is good enough for now, since this is enough to enable gray zone simulation. 250 meters is used as maximum radio range for 11 Mbps, as this is the range used by ns-2 and other simulation tools, while 500 meters is used for 1 Mbps, as this seems to be a rough general estimate for 1 Mbps. Should other radio ranges be preferred in the future, the adjustment of NEMAN to using these can be done relatively quickly.

When our gray zone is as large as this, high packet loss on the 250-500 meter links is necessary if we are to avoid getting gray zones in most of the discovered routes, effectively hindering most of the communication. The authors of NE, another MANET emulator, proposed an exponential model for the FER - Frame Error Rate, which is the amount of frames received with errors, divided by the total number of received frames. Based upon this model, we determined that our 250-500 meter links should loose about 75% of all packets. Functionality testing showed that this was enough to make OLSR favor the 0-250 meter links in many cases, thus avoiding the gray zones. Whether 75% packet loss on 250-500 meters is realistic, is hard to say, as actual packet loss varies greatly with each network card and the environment. Most likely 75% is not too high, but it might be a bit too low, as IEEE 802.11 is not meant to be used over such distances. Still, this gives us the effect of random packet loss and makes the gray zone simulation work as it should. The amount of loss is also easily configured when a different network is wanted.

These improvements to the physical and MAC layer model of NEMAN makes the emulator more realistic, which in turn makes it an even better tool for MANET application development. With these improvements, we can now revise Fig. 2.4 to show slightly higher wireless model accuracy, and the result is found in Fig. 6.1. The physical and MAC layer model of NEMAN is not realistic yet, but the functionality evaluation shows that the changes done during this thesis is a good step in the right direction. For emulation of a large MANET on a single machine, NEMAN is a very good choice, while other emulators should be considered when a highly detailed physical and MAC layer is more important than the size of the MANET. Finally, the performance of the new NEMAN Topology Manager is also satisfactory, since it can emulate more than 175 nodes on our rather outdated test machine, as well as avoiding being a larger bottleneck than the GUI. Making the physical and MAC layer models of NEMAN more accurate is still possible, even though everything runs on the same machine. In the

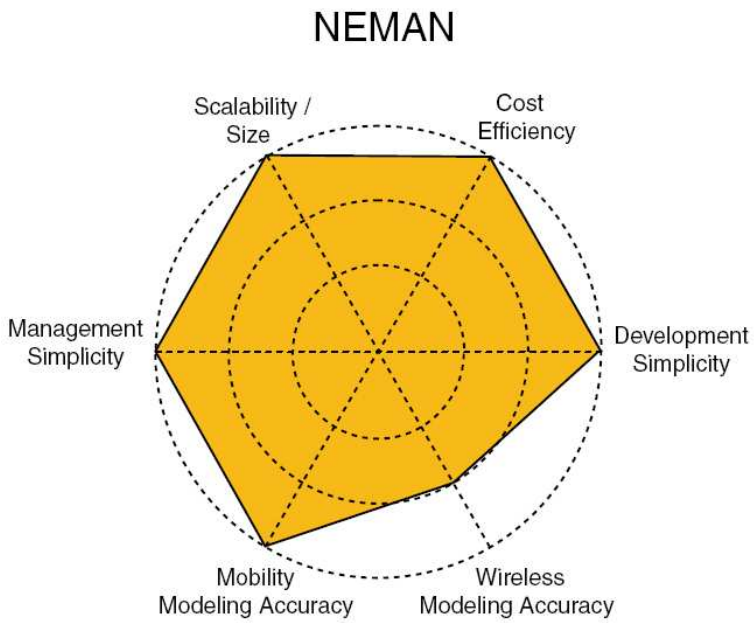

Figure 6.1: Evaluation of the new NEMAN using the criteria outlined in Sect. 2.2, modified version of figure borrowed from [12]

next section we discuss future work on NEMAN.

6.2 Future work

The main priority with NEMAN should be to improve the MAC layer model to simulate CSMA/CA on the virtual nodes of NEMAN. The results from the current traffic collision implementation is not comparable to a real world MANET, since no collision avoidance/handling algorithm is used on the nodes. In the current NEMAN implementation, every node transmits blindly, without "checking" for other nodes already using the wireless medium. The amount of collisions is therefore much larger than it would have been if e.g. CSMA/CA was simulated so that nodes only transmit when no other nodes are transmitting at the same time. Even a relatively quick implementation where collisions only occurred when two or more transmitting nodes are hidden to each others, would improve the realism greatly. Implementing optional use of RTS/CTS would also be interesting. CSMA/CA is probably the best choice for protocol to simulate on the MAC layer, as it is used in the IEEE 802.11 standards.

Different link speeds is another issue that should be investigated further, and possibly implemented on the virtual nodes. Better data, or a model, on the transmit range for each IEEE 802.11 link speed is needed if such an implementation is done. In addition, simulation of IEEE 802.11's Adaptive Rate Selection algorithm should also be added, to avoid locking each node to one link speed and range. Further details on this can be found in Sect. 3.4.

Another interesting proposal for future work would be to do a comparison between the new NEMAN and ns-2 by using the same scenario files and OLSR for routing on both. Such a comparison might help us to determine if the results from the current NEMAN implementation is comparable to ns-2 or not.

Since the current GUI, written in TcL/Tk, is a large bottleneck on the size of the scenario, especially with the original NEMAN, a new GUI written in a compiled language could be a great addition. Possibly optimizing the new Topology Manager, as well as using a faster machine, would increase the number of nodes that can be emulated by the new NEMAN. However, with the current GUI, a faster machine for the Topology Manager would still not allow us to emulate more nodes.

Finally, the effects of adding additional distance steps to NEMAN should be investigated. Using more fine grained distance values would enable us to use a different FER value on each of these steps, giving us a smoother increase of packet loss with increasing distance. The goal of such an investigation should be to determine how fine grained the distance steps could be, before the performance is lowered significantly due to the increased number of link status updates between the GUI and the Topology Manager. A further literature study should probably be done to see whether other FER models exist, and if so, compare these to the NE model when using more fine grained distance steps.

An alternative, or possibly an addition, to using more fine grained distance steps, would be to determine proper data for the transmit ranges of the different IEEE 802.11b rates for a specific setup. These range values could be used for the distance steps, possibly in combination with more fine grained values. Determining the IEEE 802.11b rates could be done through literature study, and maybe even by doing a practical experiment. Using these values with NEMAN would make it more realistic, as well as allowing us to reduce the gray zone to only cover the range between 2 Mbps and 1 Mbps. This in turn, would make the gray zone simulation more realistic, as well as usable even if the FER on the gray zone links is low.

Appendix A

CD Contents

Attached with this report is a CD containing the source code for both versions of NEMAN, and the scenario files used for all the experiments in this report. The contents of the CD:

- **neman-1.0** - The source code for the original version of NEMAN.
- **neman-1.1** - The source code for the new NEMAN version.
- **scenario** - Scenario files, the *setdest* program used to generate them, and *mkdist.pl* used adjust the scenario files for the new NEMAN.
- **libsocket** - The libsocket library.

Bibliography

[1] Daniel Aguayo, John Bicket, Sanjit Biswas, Glenn Judd, and Robert Morris. Link-level measurements from an 802.11b mesh network. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 121–132, New York, NY, USA, 2004. ACM Press.

[2] AODV-UU, Ad-hoc On-demand Distance Vector Routing, Uppsala University. [online] `http://core.it.uu.se/AdHoc/AodvUUImpl`, 2006. Visited 04.04.2006.

[3] Vaduvur Bharghavan, Alan Demers, Scott Shenker, and Lixia Zhang. MACAW: a media access protocol for wireless LANs. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 212–225, New York, NY, USA, 1994. ACM Press.

[4] Erlend Birkedal, Geir Horn, Jan Erik Johnsen, and Tore Ottersen Løkkeberg. A Comparison of AODV and OLSR Routing in Ad-Hoc networks using the NEMAN Simulator. Technical report, University of Oslo, Department of Informatics, 2005.

[5] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 85–97, New York, NY, USA, 1998. ACM Press.

[6] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626 (Experimental), `http://www.ietf.org/rfc/rfc3626.txt`, oct 2003.

[7] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501 (Informational), `http://www.ietf.org/rfc/rfc2501.txt`, jan 1999.

[8] Dell.com. Specifications - Dell TrueMobile 1180 Wireless LAN. [online] `http://premiersupport.dell.com/support/edocs/Network/P53163/en/SPECS.HT%M`, 2006. Visited 20.04.2006.

[9] IEEE 802.11, Wikipedia.org. [online] http://en.wikipedia.org/wiki/802.11, 2006. Visited 01.02.2006.

[10] INF5090, University of Oslo. [online] http://www.uio.no/studier/emner/matnat/ifi/INF5090/, 2006. Visited 19.04.2006.

[11] Phil Karn. MACA - a new channel access method for packet radio. In *In Proceedings of the 9th Computer Networking Conference*, pages 134–140, sep 1990.

[12] Matthias Kropff, Tronje Krop, Matthias Hollick, Parag S.Mogre, and Ralf Steinmetz. A survey on real world and emulation testbeds for mobile ad hoc networks. In *Proceedings of 2nd IEEE International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM 2006), Barcelona, Spain*, March 2006.

[13] James F. Kurose and Keith W. Ross. *Computer Networking, A Top-Down Approach Featuring the Internet*. Addison Wesley Longman, Inc, 2001.

[14] Weiguo Liu and Hantao Song. Research and implementation of mobile ad hoc network emulation system. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference*, pages 749–754, jul 2002.

[15] Henrik Lundgren, Erik Nordström, and Christian Tschudin. Coping with communication gray zones in IEEE 802.11b based ad hoc networks. In *WOWMOM '02: Proceedings of the 5th ACM international workshop on Wireless mobile multimedia*, pages 49–55, New York, NY, USA, 2002. ACM Press.

[16] Priya Mahadevan, Adolfo Rodriguez, David Becker, and Amin Vahdat. MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. Technical report, Dept. of Comput. Sci., University of California, San Diego, USA, http://ramp.ucsd.edu/~pmahadevan/publications/Mobinet_techrep.pdf, 2004. Visited 18.11.2005.

[17] NcFTPd Server. [online] http://www.ncftp.com/ncftpd/, 2006. Visited 10.05.2006.

[18] The Network Simulator - ns-2. [online] http://www.isi.edu/nsnam/ns/, 2006. Visited 21.03.2006.

[19] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), http://www.ietf.org/rfc/rfc3561.txt, jul 2003.

[20] Matija Pužar and Thomas Plagemann. NEMAN: A Network Emulator for Mobile Ad-Hoc Networks. In *Proceedings of the 8th International Conference on Telecommunications (ConTEL 2005)*, jun 2005. ISBN 82-7368-274-9.

[21] Saikat Ray, Jeffrey B. Carruthers, and David Starobinski. RTS/CTS-induced congestion in ad hoc wireless LANs. In *Wireless Communications and Networking, 2003. WCNC 2003.*, pages 1516–1521. IEEE, mar 2003.

[22] Saikat Ray, David Starobinski, and Jeffrey B. Carruthers. Performance of wireless networks with hidden nodes: A queuing-theoretic analysis. *Computer Communications Journal*, 28(10):1179–1192, 2005. Special issue on Performance Issues of Wireless LANs, PANs, and Ad Hoc Networks.

[23] Michael W. Ritter. The Future of WLAN. *Queue*, 1(3):18–27, 2003.

[24] Cristian Tuduce and Thomas Grosst. A Mobility Model Based on WLAN Traces and its Validation. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, pages 664–674, mar 2005.

[25] Andreas Tønnesen. OLSR daemon. [online] `http://www.olsr.org/`, 2006. Visited 21.03.2006.

[26] Wireless Frequently Asked Questions, The Center for Wireless Telecommunications, Virginia Polytechnic Institute and State University. [online] `http://www.cwt.vt.edu/faq/default.htm`, 2002. Visited 29.11.2005.

[27] Junlan Zhou, Zhengrong Ji, and Rajive Bagrodia. TWINE: a hybrid emulation testbed for wireless networks and applications. In *IEEE INFOCOMM 2006*, apr 2006.